EVALUATING THE EFFECTIVENESS OF IP HOPPING VIA AN ADDRESS ROUTING GATEWAY

THESIS

Ryan A. Morehart, Second Lieutenant, USAF

AFIT-ENG-13-M-35

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-13-M-35

EVALUATING THE EFFECTIVENESS OF IP HOPPING VIA AN ADDRESS ROUTING GATEWAY

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science

Ryan A. Morehart, B.S.

Second Lieutenant, USAF

March 2013

AFIT-ENG-13-M-35

EVALUATING THE EFFECTIVENESS OF IP HOPPING VIA AN ADDRESS ROUTING GATEWAY

Ryan A. Morehart, B.S.
Second Lieutenant, USAF

Approved:

Barry Mullins — 21 Feb 13

Dr. Barry E. Mullins, PhD (Chairman) — Date

Rusty O. Baldwin — 22 Feb 13

Dr. Rusty O. Baldwin, PhD (Member) — Date

Timothy H. Lacey — 21 Feb 13

Dr. Timothy H. Lacey, PhD (Member) — Date

AFIT-ENG-13-M-35

Abstract

This thesis explores the viability of using Internet Protocol (IP) address hopping in front of a network as a defensive measure. Network address space randomization techniques theoretically provide protection to a network by appearing to randomly change the addresses of hosts inside, presenting a challenge to an intruder attempting to break in and map the network. This research presents a custom gateway-based IP hopping solution called Address Routing Gateway (ARG) that combines previous work in this area.

ARG works as a transparent gateway in front of a network, requiring no changes to the hosts inside or out. Each ARG gateway is configured with a small amount of knowledge on one or more other gateways, allowing them to connect and pass fully encrypted and authenticated traffic amongst themselves. Connections to non-ARG networks or hosts are handled gracefully, allowing long-lived connections to exist without terminating them during IP address changes. This thesis tests the overall stability of ARG, the accuracy of its classifications, the maximum throughput it can support, and the maximum rate at which it can change IPs and still communicate reliably.

This research is accomplished on a physical test network with nodes representing the types of hosts found on a typical, corporate-style network. Direct measurement is used to obtain all results for each factor level. Tests demonstrate ARG classifies traffic correctly, with no false negatives and less than a 0.15% false positive rate on average. The test environment conservatively shows this to be true as long as the IP address change interval exceeds two times the network's round-trip latency; real-world deployments may allow for more frequent hopping. Results show ARG capably handles traffic of at least four megabits per second with no impact on packet loss. Fuzz testing validates the stability of ARG itself, although additional packet loss of around 23% appears when under attack.

Acknowledgments

I would like to thank my wife for putting up with late nights, missed phone calls, and many terrible revisions. Her patience has been astounding. Without her support this would have not been possible. Also, thanks to Dr. Mullins for giving me the freedom to direct my own work, yet always being willing to provide advice when needed.

Ryan A. Morehart

Table of Contents

Page

Abstract iv

Acknowledgments v

Table of Contents vi

List of Figures ix

List of Tables x

List of Acronyms xii

I. Introduction 1

1.1 Motivation 1
1.2 Goals and Limitations 2
1.3 Thesis Overview 3

II. Background 4

2.1 Network Routing 4
2.1.1 Internet Protocol (IP) Routing 4
2.1.2 Network Address Translation 6
2.1.3 Ethernet and Address Resolution Protocol (ARP) 8
2.2 IP Hopping in Detail 9
2.2.1 End Point Hopping 11
2.2.2 Gateway hopping 13
2.3 Data Security 15
2.3.1 Hashing 15
2.3.2 Encryption 16
2.3.3 Authentication 17
2.3.4 Combining for Full Effect 17
2.4 Time-Based One-Time Password (TOTP) 18
2.5 Previous Implementations 19
2.5.1 BBN's Dynamic Network Address Translation (DYNAT) 19
2.5.2 Sandia Dynat 20
2.5.3 Applications that Participate in their Own Defense (APOD) 20
2.5.4 Network Address Space Randomization (NASR) 21

Page

2.5.5 Network Address Hopping (NAH) . . . 22
2.5.6 Transparent Address Obfuscation (TAO) . . . 23
2.6 Summary . . . 23

III. Implementation . . . 24

3.1 Requirements . . . 24
3.2 Architecture Overview . . . 25
3.3 Components . . . 27
3.3.1 Director . . . 28
3.3.2 Hopper . . . 30
3.3.3 Network Address Translator . . . 32
3.4 Address Routing Gateway (ARG) Protocol . . . 33
3.5 Summary . . . 36

IV. Methodology . . . 37

4.1 Problem Definition . . . 37
4.1.1 Goals and Hypothesis . . . 37
4.1.2 Approach . . . 38
4.2 System Boundaries . . . 38
4.3 System Services . . . 39
4.4 Workload . . . 41
4.5 System Parameters . . . 42
4.6 Evaluation Technique . . . 42
4.7 Performance Metrics . . . 44
4.8 Experimental Design . . . 45
4.9 Summary . . . 48

V. Results and Analysis . . . 50

5.1 Basic Tests . . . 50
5.1.1 Valid Packet Loss . . . 50
5.1.2 Invalid Packet Loss . . . 54
5.2 Minimum Hop Interval . . . 56
5.3 Maximum Packet Rate . . . 61
5.4 Fuzzing Test . . . 65
5.5 Overall Analysis . . . 68
5.6 Summary . . . 69

Page

VI. Conclusions and Recommendations 70

6.1 Research Conclusions 70
6.2 Research Impact 70
6.3 Future Work 71
6.3.1 IPv6 Support 71
6.3.2 Fragmentation Support 71
6.3.3 More Extensive Malicious Testing 71
6.3.4 More Intelligent NAT 72
6.3.5 Integration with Other Defenses 72
6.3.6 Latency Compensation 72
6.4 Summary 73

Appendix A: IP Packet Structure 74

Appendix B: ARG Protocol 75

Appendix C: ARG Testing 80

Appendix D: ARG Building and Configuration 82

Appendix E: Traffic Generators 89

Appendix F: Results Processor 94

Bibliography 99

List of Figures

Figure Page

2.1 IP routing example network 5

2.2 ARP exchange example 9

2.3 IP hopping example network 10

3.1 ARG conceptual network layout 26

3.2 Packet sent between gateways when hop interval is half the latency 28

3.3 ARG director flow 29

3.4 ARG incoming packet validation process 32

4.1 ARG System Under Test (SUT) diagram 39

4.2 ARG test network layout overview 43

4.3 Experiment traffic flow directions and protocols 49

5.1 Basic tests, raw valid packet loss 51

5.2 Basic tests, valid packet loss means and confidence intervals 52

5.3 Hop interval tests, packet loss of User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) traffic between ARG networks 56

5.4 Hop interval tests, scaled view of packet loss between ARG networks 57

5.5 Time synchronization process, including ARPs 59

5.6 Hop interval tests, packet loss of TCP traffic between ARG networks 62

5.7 Hop interval tests, packet loss of externally-bound traffic 64

5.8 Packet rate tests, clustered loss verses throughput 66

5.9 Packet rate tests, Tukey test against clustered data 67

List of Tables

Table		Page
2.1	IP routing example: Router A table	5
2.2	Network Address Translation (NAT) table example	7
2.3	Hashing example	15
3.1	Information hopper module maintains on other ARG gateways	31
3.2	ARG NAT table example	33
3.3	ARG packet data	34
3.4	ARG message types	35
3.5	ARG message security summary	36
4.1	Factor levels for basic tests	46
4.2	Factor levels for throughput tests	46
4.3	Factor levels for minimum hop interval tests	47
4.4	Factor levels for fuzz tests	47
5.1	Basic tests 1-4 packet rejection reasons	53
5.2	Basic tests, packet loss of invalid traffic	54
5.3	Basic tests 5-8 packet rejection reasons	55
5.4	Hop interval test loss reasons	63
5.5	Packet rate clusters	65
A.1	IP packet structure	74
B.1	Data in ARG `WRAPPED` message	75
B.2	Data in ARG `PING` message	76
B.3	Data in ARG `CONN_REQ` and `CONN_RESP` messages	76
B.4	Data in ARG `TRUST_DATA` message	77
C.1	Test run `tcpdump` calls	80

Table Page

E.1 gen_traffic.py command-line parameters 92

F.1 `process_run.py` command-line parameters 95

List of Acronyms

Acronym	Definition
AES	Advanced Encryption Standard 16
APOD	Applications that Participate in their Own Defense 20
ARG	Address Routing Gateway 1
ARP	Address Resolution Protocol 8
CI	confidence interval 50
COTS	Commercial Off-The-Shelf 21
CPU	Central Processing Unit 55
CUT	Component Under Test 37
DARPA	Defense Advanced Research Projects Agency 20
DHCP	Dynamic Host Configuration Protocol 21
DOS	Denial of Service 55
DYNAT	Dynamic Network Address Translation 19
HMAC	Hashed Message Authentication Code 15
HOTP	HMAC-Based One-Time Password 18
HTTP	Hypertext Transport Protocol 19
IDS	Intrusion Detection System 1
IP	Internet Protocol 1
IPsec	IP Security 17
IPv4	IP version 4 4
IPv6	IP version 6 4
ISP	Internet Service Provider 6
IV	Initialization Vector 76
Kbps	kilobits per second 45

Acronym	Definition
MAC	Media Access Control 8
Mbps	megabits per second 38
NAH	Network Address Hopping 22
NASR	Network Address Space Randomization 1
NAT	Network Address Translation 6
PCAP	Packet Capture 43
RAM	Random Access Memory 44
RSA	Rivest, Shamir, and Adleman 16
RTT	Round-Trip Time 45
SHA	Secure Hash Algorithm 15
SUT	System Under Test 37
TAO	Transparent Address Obfuscation 23
TCP	Transmission Control Protocol 14
TOTP	Time-Based One-Time Password 4
UDP	User Datagram Protocol 34
VLAN	Virtual Local Area Network 44
VPN	Virtual Private Network 20
WAN	Wide Area Network 60

EVALUATING THE EFFECTIVENESS OF IP HOPPING VIA AN ADDRESS ROUTING GATEWAY

I. Introduction

1.1 Motivation

Traditional network defenses consist of largely static tools. Firewalls and Intrusion Detection Systems (IDSs) form the backbone of protection in most information technology shops. Despite extensive work and research into these systems, attackers still routinely break into networks and bring down critical systems, exfiltrate data, and establish footholds for future actions. In an effort to combat this, interest has increased in more active defense mechanisms, such as reputation and trust-based security [NV09] and Network Address Space Randomization (NASR) [APWJ03, SSH05]. This thesis focuses on the latter in a large corporate network setting.

At a high level, the concept of NASR is simple: rather than a system sitting on a single Internet Protocol (IP) address, it changes addresses rapidly, hopping amongst a set of IP addresses assigned to it. Normally an attacker wishing to target a given network is capable of gaining intelligence through simple scanning, checking each IP inside the network and then checking each port on the active IPs to see what services are available. With this knowledge, the attacker can often find an entrance into the network. IP hopping mitigates this issue by making it difficult to probe the network in the first place and quickly invalidating any network map that the attacker does manage to generate; even if they do manage to look a system's internal IP at one point in time, the address will change moments later.

1.2 Goals and Limitations

This thesis proposes an IP address hopping system called Address Routing Gateway (ARG). It incorporates many of the features of previous address-hopping schemes, with an eye on the specific needs of the military and any other large, geographically diverse corporation. In this context each of the existing systems presents drawbacks that ARG attempts to avoid. Additionally, the design of ARG is intended to allow its future integration with a traditional IDS and honeypot, potentially gaining additional insight into an attacker's behavior.

The geographically diverse military network demands high availability, reliability, and security over any network. Additionally, ARG must support transport over the commercial Internet. In light of these and other requirements, this thesis examines several questions with regards to IP hopping and ARG.

- Does ARG classify traffic correctly? What percentage of false positives (valid packets blocked) and false negatives (invalid traffic allowed through) does it introduce?
- What is the maximum packet rate ARG can support?
- What is the minimum hop interval—the time between ARG's IP changes—that is supportable? How does latency affect this?
- Is ARG stable when presented with corrupt, malformed, and/or replayed packets?

Each question is tested in a separate series of experiments. Direct measurement on a physical network is used for all results. Nodes on the test network represent the types of hosts found on a typical, corporate-style network. These include trusted hosts inside trusted networks which communicate freely, internal and external servers that must be accessible to hosts inside these trusted networks, and malicious hosts outside the networks. ARG sits

in front of the trusted networks and provides network address space randomization to the internal hosts.

1.3 Thesis Overview

This chapter introduces the research, goals, and limitations of the work in this thesis. Chapter 2 covers foundational background topics, concepts, and research. Chapter 3 discusses the design of ARG, including design requirements, architecture, and the network protocol. Chapter 4 walks through the test methodology used in this thesis, while Chapter 5 presents results and analysis of these tests. Finally, Chapter 6 provides a concluding discussion of this research and presents possible areas of future work.

II. Background

This chapter provides an introduction to the technology behind IP hopping and many of the previous efforts in this area. Section 2.1 describes how routing works at various points in a network. Section 2.2 covers IP address space randomization—referred to as "IP hopping" in this thesis—and two possible approaches. Sections 2.3 and 2.4 provide descriptions of two critical systems behind the implementation this thesis presents, encryption and Time-Based One-Time Password (TOTP). Finally, Section 2.5 examines previous efforts in IP address hopping.

2.1 Network Routing

2.1.1 IP Routing.

This thesis assumes that the reader has a familiarity with how IP works. However, several aspects of this protocol are critical to the functioning of the system described later and are detailed here.

IP packets are routed from system to system based on the destination IP address contained in their header. Appendix A displays the format for both IP version 4 (IPv4) and IP version 6 (IPv6) packets. IPv4 uses 32-bit addresses to uniquely identify each system on the public Internet. These addresses are typically represented in a "dotted quad" format: for example, 74.125.228.36. IPv6 uses 128-bit address, typically represented in hexadecimal and separated by colons, along the lines of fe80::baf6:b1ff:fe1b:b4a1.

Regardless of which version is in use, high-level routing remains conceptually the same. Routers maintain a table of IP addresses, masks, and the interfaces associated with each. When a packet is received, the router consults this table and decides what interface to send the packet out on based on the most specific entry. For example, in the network

shown in Figure 2.1, the laptop with IP 10.5.0.25 wants to send a packet to 172.100.10.3. When 10.5.0.25 sends its packet, the following sequence of events occurs:

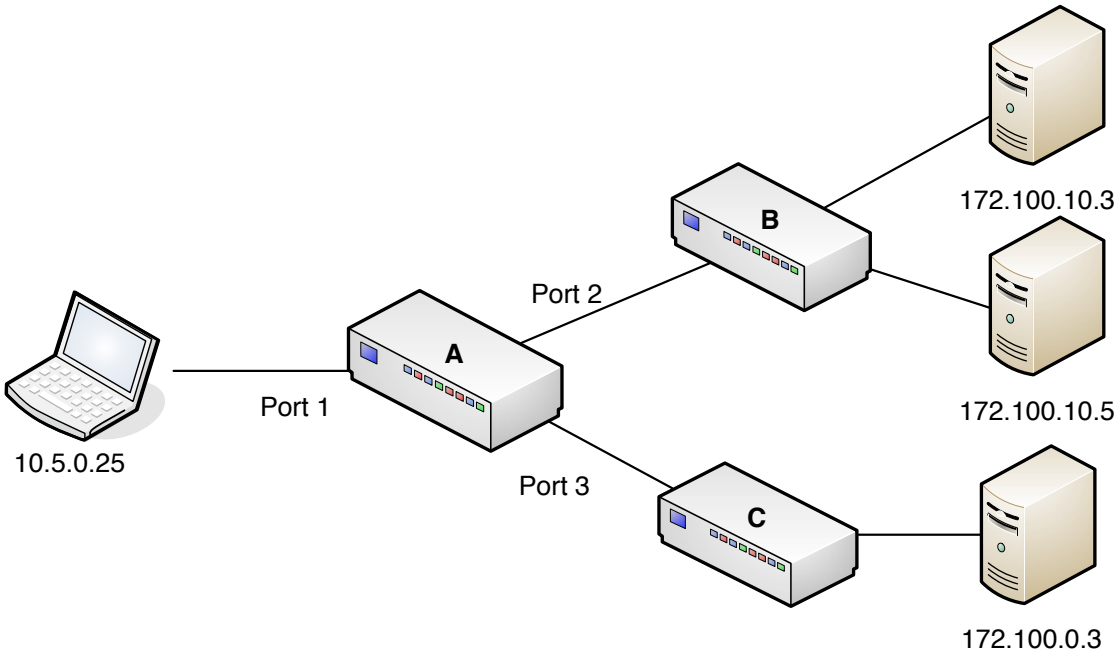


Figure 2.1: IP routing example network

1. The packet leaves 10.5.0.25 and Router A receives it on its interface Port 1.

2. Router A compares the packet's destination IP (172.100.10.3) to its table, which looks like Table 2.1.

Table 2.1: IP routing example: Router A table

	IP	**Mask**	**Interface**
1	10.5.0.25	255.255.0.0	Port 1
2	172.100.10.0	255.255.255.0	Port 2
3	172.100.0.0	255.255.0.0	Port 3

3. Router A determines the IP matches best with entry 2, which instructs the router to forward the packet via Port 2.

4. Router B receives the packet and does a similar lookup, forwarding it out on the port to 172.100.10.3.

5. 172.100.10.3 receives the packet.

This scheme allows a router to direct packets without having to know every individual IP; they only need to know broad address ranges. In the example, Router A possesses no knowledge of how to get the packet directly to 172.100.10.3, but it does know which direction to send it. This becomes exponentially more useful on larger networks. A corporation's network, for instance, may contain hundreds or thousands of addresses, but the routers directing packets to them need only have one entry in their table to correctly route packets. The internal routers of the corporation are then in charge of further routing.

This limited-knowledge architecture allows IP hopping to work. As long as the tables of the routers outside the network contain the correct IP ranges, the systems inside are free to change addresses as frequently as they want and handle internal routing any way they desire.

2.1.2 Network Address Translation.

Network Address Translation (NAT) is a core technology behind many modern home and corporate networks, allowing many systems to connect to the Internet yet appear to come from a single external IP address. A typical home network, for instance, might have the external IP address 184.58.31.151 assigned to it by their Internet Service Provider (ISP) yet have five systems—laptops, desktops, phones—inside with IPs like 192.168.0.103, 192.168.0.50, and 192.168.0.1. As these systems send requests out, the router changes the source IP (and port) for packets to the external IP address. As responses come back

from the Internet, the router does the opposite, changing the destination of the packets from the external IP to the internal IP of the original requester.

To do this, routers must maintain a NAT table. This table consists of the source and destination information as well as a new port number, allowing the router to consistently transform traffic in both directions. For example, a router might have a table like the one shown in Table 2.2.

Table 2.2: NAT table example

	Int IP	**Int Port**	**Remote IP**	**Remote Port**	**Ext Port**
1	192.168.0.103	3547	74.125.225.69	443	50003
2	192.168.0.103	8751	207.109.73.34	80	42630
3	192.168.0.112	30452	4.27.2.253	80	53920

This small table shows three different connections in progress. The router created each entry the first time an internal system sent a packet to the remote (Internet) system, for each set of internal and remote IPs and ports. In addition, the router assigns an external port for each connection to allow it to determine the destination of incoming packets.

In the future, when the router receives an outgoing packet (from the internal host to the external), it begins by consulting its table to find a match based on the first four values in the table. Based on the table entry, the router changes the source IP of the packet to the external IP assigned to it by the ISP and the source port to the external port given in the table (if it does not find one, it creates a new one). When the router receives a packet from the Internet, it checks for a match in the table based on the remote IP, remote port, and external port. If it finds one, it alters the packet's destination information to the internal IP and port; if it does not find a match, it drops the packet.

This system serves two purposes. First, as the number of systems on the Internet has increased, IPv4 addresses have become a limited resource, with their 32-bit length limiting the number of possible addresses to around four billion. NAT allows an organization to only own a single address yet serve many systems behind it. Second, NAT inherently acts as a simple stateful firewall [ARMT06]. In order for an outside system to send packets to an internal host, the internal host must initiate the connection, allowing the router to create the table entry.

2.1.3 Ethernet and Address Resolution Protocol (ARP).

Many local networks use Ethernet for the first and last leg of network travel to actual hosts. In a flat network, where local machines are connected together via switches or hubs, IP routing is not typically used. Instead, packets are directed to the correct recipient via physical identifiers known as Media Access Control (MAC) addresses. Packets sent on an Ethernet network are wrapped in an Ethernet frame, which specifies the MAC addresses of the sender and receiver.

When a packet is first created, however, the host system only knows the destination IP. Before the packet can be sent out, the sender must determine the destination MAC address. This is done through an ARP request, a process illustrated in Figure 2.2.

As shown, the sender first asks on the network who has the destination IP. Every host on the network (typically) hears the request, but only the host with that IP responds, sending back an ARP response with their MAC address to the requester. With the destination MAC now in hand, the original sender can construct the Ethernet frame for their IP packet and send the data on its way. The sender caches the physical address of the other machine for a short time to avoid repeating the ARP request too frequently.

To send packets beyond the local network, hosts use a gateway system, usually a router. They transfer the packet to the gateway via Ethernet, then the gateway directs the

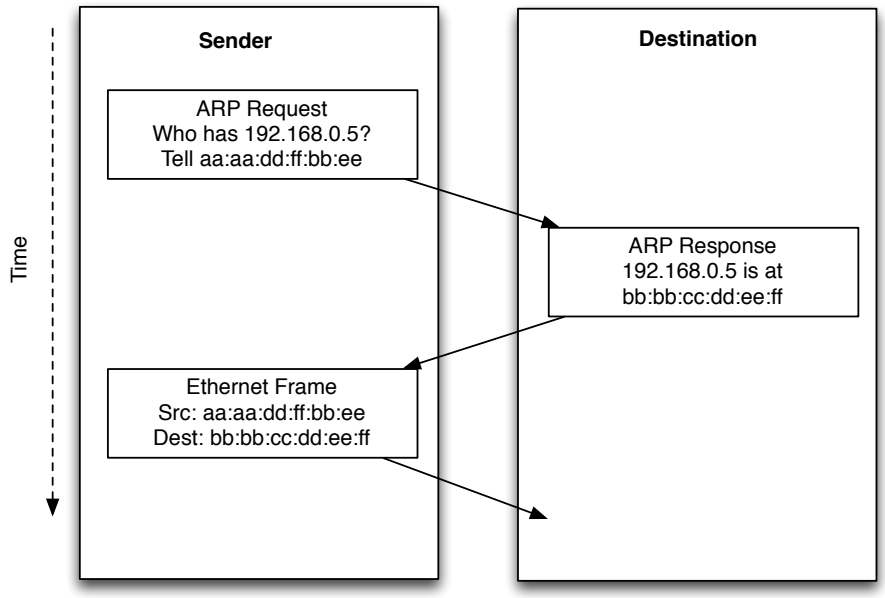


Figure 2.2: ARP exchange example

packet further. Ethernet and ARP are not used directly by local hosts to reach hosts beyond the gateway.

2.2 IP Hopping in Detail

Address hopping is a simple concept at a high level: take the basic identifiers of a network and mutate them in a way that only authorized systems understand and continue to use for communication. In this thesis, hopping focuses on changing the IP addresses the "hopping" systems use. In some ways this is similar to frequency hopping, where senders and receivers change the frequency in use in a synchronized manner to avoid interference, jamming, and eavesdropping [MKR+04]. In the same way, changing the IP addresses of packets makes it difficult for an adversary to correlate sniffed traffic with individual machines and even more difficult to probe into the network to enumerate hosts. Other methods of dynamic network reconfiguration exist, but address hopping may have the greatest potential for obstructing network reconnaissance efforts [Rep08].

In trying to actually implement such a system, however, several issues arise. The network Figure 2.3 illustrates is used to aid the following discussion. There are two main networks *A* and *B* that wish to communicate freely. They are connected via the Internet and are assigned the displayed IP ranges. Each of these has a few friendly end nodes (*A1*, *A2*, *B1*, etc.) behind a main router (*AR* and *BR*). Additionally, network *B* has a potentially rogue client inside it named *M1*. Outside of those two networks is the friendly *C1* node, who has an interest in at least occasionally communicating with nodes inside networks *A* and *B*, and malicious *M2*, who wants access to said networks. The details of the routes between them and *A* and *B* are inconsequential.

Note that for the sake of this discussion non-routable IPv4 addresses are used. This is done merely for convenience and readability, everything applies to IPv6 as well unless otherwise noted.

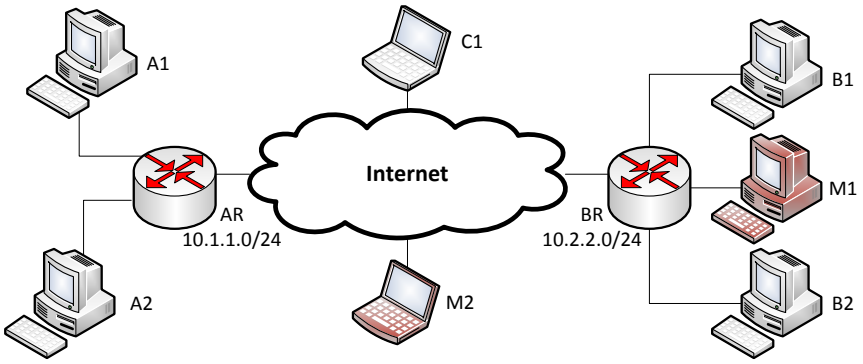


Figure 2.3: IP hopping example network layout. Red nodes are malicious.

There are two basic ways to deploy IP address hopping on this network: each end point hops individually or the network gateways transform incoming and outgoing packets. Both options and their strengths and weaknesses are discussed.

2.2.1 End Point Hopping.

For the example network, end point hopping means that all nodes behind *AR* and *BR* (*A1*, *A2*, *B1*, etc.) change addresses on a periodic basis, independent of one another. Despite the apparent simplicity of this setup, several questions must be answered.

First, how do the nodes keep track of one another? If every node knows about all the others, then scalability might become an issue, as every client presumably has to maintain some amount of data on fellow hopping clients to determine the IP each one possesses at any given time. It may be possible to devise a scheme where this flaw is mitigated by having all clients hop using the same secret and they each know just the broad IP ranges where fellow hoppers reside (i.e., the *AR* nodes know that 10.2.2.0/24 is a network they talk to with hopping), but this accentuates the next question: how do clients choose their hops?

Each node must know some secret from which IP addresses are generated, hopefully in a manner which appears random to outsiders yet is predictable for those with the secret. The secret combined with the algorithm used to generate IP addresses results in what this thesis refers to as a "hopping pattern." The most obvious approach is to share a single secret amongst all nodes in a give "hopping network." In the example network, this would mean `A1`, `A2`, `B1`, and `B2` all possess the same secret. While straightforward, this does have the potential flaw of revealing too much information to an eavesdropper though: with a large number of nodes all following a hopping pattern based on the same key, it may be easier to deduce the secret [Sha49].

Giving each node a separate secret would solve this issue. However, each client now has to maintain even more information on every other client. For a small number of nodes this is not a problem, but if the intention is to scale to thousands or tens of thousands of clients problems with processing and key distribution may arise.

Just as importantly, however, is the question of how nodes coordinate their IP address choices. As discussed in Section 2.1, IP routing requires addresses follow a largely

hierarchical setup. Thus, if *A* owns the network address range 10.1.1.0/24, then all of its nodes must fall within the address range of 10.1.1.1 through 10.1.1.255. This means that when each node hops it must remain within the valid range of the network to which it connects *and* that the address it chooses must not already be taken by another node. Given enough nodes in a subnet, a conflict is quite possible, leading to unpredictable network behavior. For example, in the example network *A* there are 255 available IP addresses. As shown by the birthday paradox, there is a 50% chance (on every hop) of having an IP address conflict with only 20 hosts, assuming uncoordinated random hops [Buc04]. On a larger network with 65,535 possible IPs (a /16 network), only 302 hosts are needed to exceed a 50% collision probability.

The easiest solution to this problem is to give each node a unique, non-overlapping address range in which to hop. This requires no on-going communication with other nodes to work well and has the distinct advantage of working properly with existing defenses (i.e., a firewall with special rules for a host can point to the range for that host, rather than an individual IP). This setup requires a large enough address space to make hopping beneficial. If a given node only has five possible addresses in which to hop, for instance, it becomes trivial for an attacker to just keep trying a single IP address until the node returns to it. IPv6 would alleviate this problem, as the Internet Engineering Task Force recommends the allocation of a /64 address space (2^{64} addresses) to every link [SWLX02], but IPv4 with its more limited address space would not allow this flexibility and, unfortunately, the reality of current networks mandates support for IPv4 [CGKR].

With enough coordination between nodes everyone could share the same address space. A fair amount of work would need to be put into an algorithm and protocol to synchronize the hosts' address changes, but this is not insurmountable. This does break most special cases in firewalls unless the firewall is either made aware of the IP hopping system or hosts that need special rules do not change addresses at all.

Despite the negatives given above, end point hopping does have advantages. First, scalability issues lie more in storage space and key lookup speed than actual computation, as every node only has to perform packet transformations for their own incoming and outgoing packets. Additionally, many IP hopping schemes incorporate encryption, which may consume a relatively large proportion of processing time [KFLD01, MPS+02]. Distributing this load may be beneficial. Second, end point hopping protects clients from probing no matter where the adversary is in the network. For example, as long as *M1* in our example network lacks the hopping key, they have no more of an advantage in scanning any of the *A* or *B* nodes than *M2*, who is outside the network. Finally, end point hopping comes with the ability for individual nodes such as *C1* to easily connect to the main hopping network without the need to develop new software or deploy extra hardware. This is in distinct contrast to gateway-based hopping, which by definition uses a separate system to handle hopping, as Section 2.2.2 discusses.

2.2.2 Gateway hopping.

The alternative to end point hopping is to move the hopping to network gateways. In such a scheme, networks are placed behind gateways that alter all traffic passing through them appropriately. What "appropriately" means varies with every implementation, but in one way or another, a gateway outside of the actual end points alters the IP traffic to make it appear as though the systems inside are changing IP addresses. Referring back to the example network in Figure 2.3, gateways *AR* and *BR* would be in charge of these packet transformations, altering traffic from the clients inside their networks (e.g., *A1*, *A2*, etc.) to outside hosts. The hopping secrets stored on end points in end point hopping are now stored at the gateway, with each gateway getting its own unique set of keys; the gateways typically lack precise knowledge of the end points they protect.

Nodes inside the network may or may not have knowledge of the hopping. In most instances the hopping occurs with no modification of the end points and is largely

transparent [AA06]. The need to only deploy a small number of gateway systems, rather than altering every client system, gives gateway-based hopping an advantage over end point-based on larger networks. Applying software and/or hardware changes to every system is costly in terms of both time and manpower. Even more significantly, legacy systems running older operating systems might need custom solutions, increasing the cost of deployment.

The most common observable side effect of gateway hopping (beyond the latency associated with the additional processing) is Transmission Control Protocol (TCP) connection dropping. Because TCP depends on IP addresses and ports numbers to identify on-going connections, any alterations to this information would traditionally kill the connection [AAMA07]. This is a problem also faced by end point hopping schemes, but is more easily corrected because the individual machines know the state of connections and can correct appropriately. With gateway hopping, however, the situation may require more significant state tracking and packet inspection at the gateway.

Because of this required state tracking and the need for a single system to alter all traffic in and out of an entire network, gateway-based hopping presents a possible performance problem. With enough traffic or individual nodes behind a gateway, it may be possible to overload the gateway, leading to dropped packets and failing connections. Depending on gateway design this may be avoided; some studies show an additional CPUs load of around 10% even when encryption is applied, when compared to standard routing [AA06]. While noticeable, this increase does not cause immediate issues. Additionally, impact would likely be lower with modern processors, due to the introduction of hardware encryption on-chip [BBGR09].

Finally, gateway hopping may have more difficulty accommodating individual clients connecting to the network. In *C1*'s case, for instance, if it wanted to be able to talk to *A1*, it would need to obtain the appropriate hopping information from *AR* and alter its own

traffic accordingly. There are several techniques to make this work, but all of them require more work than an end-point hopping system would, simply because end-point hopping is already designed around the concept of individual nodes connecting together.

2.3 Data Security

2.3.1 Hashing.

Cryptographic hash functions take a given sequence of bytes and return a fixed-length string of bits representing that data. While their output is not unique for every input, cryptographic hashes attempt to make it infeasible to generate two messages with the same hash or to modify an input and get the same hash. As illustrated in Table 2.3, these properties allow the use of hashes to verify data because even small changes result in different output.

Table 2.3: Hashing example

	Input	**Output (SHA-1)**
Original	The quick brown fox jumped	c950af1b07223c7d8590538189b3bcd9f4e08c6c
Changed	The quick br***a***wn fox jumped	97c592d5c0d991b91c68edb3941b1bb075a97f56

Here even a small change (*o* to *a*) resulted in a significantly different output. If the sender gave both the data and the hash to a recipient, the recipient would be able to repeat the hash on the data and verify that their hash matches the one the sender gave them. Hash algorithms, like the Secure Hash Algorithm (SHA) family used in this example, are widely published, so anyone can produce a valid hash of any data they want. This means that hashes alone do not prevent against malicious modification: an attacker can modify data, then produce a new hash to match. To counter this problem, either a digital signature or a Hashed Message Authentication Code (HMAC) must be used, as covered in Section 2.3.3.

2.3.2 Encryption.

Encryption is a crucial component of most hopping systems. Additionally, encryption forms the basis of digital signatures, a means of verifying that a given message—a single packet, in most contexts here—came from where the receiver believes. It is not important for the purposes for this thesis to understand the math behind encryption, but a few concepts are helpful.

Encryption comes in two major flavors, symmetric and asymmetric. Symmetric encryption uses the same key for encryption and decryption and, in many cases, is relatively fast. Newer CPUs, such as Intel's Core processors, include hardware instructions for the commonly used Advanced Encryption Standard (AES) algorithm, increasing symmetric encryption's speed even further [Gue10]. Because anyone with the encryption key can decrypt the data, participants must establish their shared secret in a secure way.

Asymmetric encryption—also referred to as public-key encryption—provides a way to exchange secure data *without* having to establish a secret key in advance. In asymmetric encryption a public key, known to everyone, and a corresponding private key, known only to one participant, are used. If a sender wants to transmit data to a specific receiver securely, they encrypt the data with the receiver's public key. When the receiver gets the transmission, they decrypt it with their private key. Because no one else knows the private key of the receiver, it is impossible for anyone else to decrypt the data, even though the encryption key is widely known. In exchange for the robustness and openness of this encryption style, asymmetric encryption algorithms tend to be slow when compared to symmetric encryption.

Asymmetric and symmetric encryption are typically used together in network communications. When a connection is first established, a public-key encryption scheme such as Rivest, Shamir, and Adleman (RSA) determines the shared symmetric key, then all future data is encrypted with a symmetric algorithm like AES [CS03]. This hybrid

approach provides the best of both worlds: no need to establish a shared secret in advance and speed during longer communications.

2.3.3 Authentication.

Authenticating that a given sequence of bytes actually comes from whom they say they do can be done through either a HMAC [KBC97] or a digital signature [JK03]. These algorithms are based on symmetric and asymmetric encryption, respectively, and therefore share the same strengths and weaknesses. Both algorithms are widely used today in protocols like IP Security (IPsec) [MG98] and user-level data like email [RT10].

HMAC works through a series of multiple hashes with the data and an encryption key. Both sender and receiver have the same shared symmetric key, allowing them each to do the HMAC process independently. The sender includes their computation of the HMAC with the original data, then the receiver calculates their own HMAC and ensures it matches the one sent with the message. If it does, the receiver knows the message is unchanged and comes from someone with the correct shared key.

For a digital signature, the sender encrypts the data's hash with their own private key. The recipient verifies the authenticity of the message by performing their own hash of the data and decrypting the signature with the sender's public key. If the calculated hash and the decrypted hash match, the sender must be who they claim to be, as only that sender has access to the correct private key.

2.3.4 Combining for Full Effect.

By combining encryption and authentication, two parties can communication with confidentiality and integrity. For public key encryption, a sender signs the message with their own private key, then encrypts the message with the public key of the recipient [An01]. The recipient decrypts the message with their private key, then verifies the signature with the public key of the sender.

For symmetric encryption, the opposite order is used. First the sender encrypts the data with one symmetric key, then adds an HMAC of the ciphertext [BN00]. The receiver computes the HMAC of the cipher text and verifies the attached one to ensure the integrity and authenticity of the message. The receiver then decrypts the data.

2.4 Time-Based One-Time Password (TOTP)

The system discussed in this thesis relies heavily on values that are unpredictable to an outsider but calculable for anyone with the secret key. The algorithm behind these values is Time-Based One-Time Password (TOTP) [MMPR11], which in turn relies on the HMAC-Based One-Time Password (HOTP) algorithm [MBH+05]. Both of these algorithms are frequently used in two-factor authentication systems employed by banks and other websites, with a smartphone app or key fob displaying the current password [AZEH09].

HOTP utilizes a key of at least 128-bits and a "counter" to produce its output. These are passed to a SHA-based HMAC, which produces a 20-byte string. It then applies a "truncation" process to the HMAC, returning a four-byte output [MBH+05]. If both sides of an exchange know the key and the current counter, they are able to independently produce the same value.

The counter for HOTP is set based on some event determined by the system designer. In the most literal case, the usage of a given output for authentication causes an internal counter to be incremented. In a situation like this, both sides of the HOTP (the sender and the receiver) must keep their counters in-sync or the two will produce differing values and need to be resynchronized.

To work around the desynchronization problem, time may instead be used as the basis of the counter. RFC 6238 defines TOTP as a simple extension of HOTP where the counter becomes the current time divided by a configured time step [MMPR11]. This change allows

all parties interested in a given TOTP to produce the same value as long as they keep their clocks relatively similar (within one time step).

2.5 Previous Implementations

2.5.1 BBN's Dynamic Network Address Translation (DYNAT).

BBN Technologies' 2001 paper entitled "Dynamic Approaches to Thwart Adversary Intelligence Gathering" tests the hypothesis that "dynamic modification of defense structure improves system assurance" [KFLD01]. In the paper they lay out a custom address space randomization solution called Dynamic Network Address Translation (DYNAT).

A series of red-team tests against their DYNAT are used to test if the addition of IP hopping decreases an adversary's ability to map the network. The experimentation confirms BBN's hypothesis: DYNAT does increase system assurance because the adversary's work greatly increases in comparison to static networks. Even when the red team receives detailed knowledge of the DYNAT's operation, the adversary could not identify a critical server in an enclave with DYNAT active [KFLD01].

BBN's implementation focuses on individual clients connecting to a server collection through a DYNAT gateway on the server end. This gateway transforms incoming and outgoing packets between "true" host identification information—e.g., the actual IP address and port number of a server inside the enclave's network—and values which vary based on a pre-shared key and time. On the client side, a "DYNAT shim" sits in the network stack and does the same thing, transparently allowing client applications to work with the server enclave. Additionally, DYNAT applies encryption to all traffic for confidentiality.

BBN's experiments also found that encryption of the packets is crucial, as the attackers can trivially sniff the traffic to find important servers, even if they do not know the real IP address or port of the target. For example, an attacker might see a packet containing an Hypertext Transport Protocol (HTTP) response and thus learn an active IP and port for a

web server, even if probing for the server is impossible. While this information remains valid for a limited period, it may give enough time for the attacker to compromise the internal network.

2.5.2 Sandia Dynat.

Sandia National Labs' 2001 final report on their extensive work in the "dynat" field examines virtually every detail in IP hopping systems, from how hopping is synchronized to where in the network it is implemented [MPS+02]. This paper points to many of the important issues that must be considered when implementing or deploying an IP hopping tool.

Of note are Sandia's recommendations on the location of the deployment of a gateway-based DYNAT. In order to avoid interference with existing firewall rules—particularly ones with a stateful firewall—a hopping gateway must be deployed beyond the current system. Likewise, for gateway-based Virtual Private Networks (VPNs), there is often a static IP requirement to allow for authentication [MPS+02], so an IP hopping gateway must also lie beyond the VPN concentrator, closer to the public-facing side of the network. Essentially, the hopping gateway should be the last system before each network connects to the outside world [MPS+02].

The Sandia report also provides significant insight into the interaction of a DYNAT with IPsec and strongly suggests a combination of the two. First, the encryption from IPsec avoids the ineffectiveness of address space randomization if the packets can be trivially sniffed for information, as BBN's work reveals [KFLD01]. Second, a DYNAT strengthens IPsec, as the hopping gateway can quickly reject invalid packets based on invalid source and destination identifiers, rather than forcing IPsec to perform expensive HMAC computations and/or encryption. However, the report also warns that the use of IPsec with a hopping gateway can reduce some aspects of DYNAT's access control because more identifiers are encrypted and unusable.

2.5.3 Applications that Participate in their Own Defense (APOD).

BBN continued work in the dynamic network address translation field and proposed another IP hopping implementation in 2003 as part of the Defense Advanced Research Projects Agency (DARPA) Applications that Participate in their Own Defense (APOD) project [APWJ03]. This system is a refinement of their previous DYNAT, featuring a NAT gateway sitting either on the server host itself or on a gateway into the network.

The primary differences between APOD and the previous DYNAT relates to implementation. Whereas the BBN DYNAT is a very specialized solution, APOD employs standard Commercial Off-The-Shelf (COTS) utilities, such as Linux's iptables, to perform much of its work. APOD's researchers propose the use of a NAT gateway on the networks of both the client and the server, rather than a gateway only at the server side and/or specialized software on each end point.

2.5.4 Network Address Space Randomization (NASR).

The 2005 Network Address Space Randomization (NASR) project is an IP address hopping system designed to defeat hitlist-based worms [AAMA07]. These worms spread to pre-collected lists of IP addresses that are likely vulnerable and typically propagate much faster than traditional worms that target random IPs. To fight this, NASR causes the pre-built hitlists to decay by changing IP addresses on a periodic basis. When the addresses change, the worm's lists are now inaccurate, giving them fewer exploitable targets.

The most unique aspect of this research is the use of Dynamic Host Configuration Protocol (DHCP) to force the changes. Through the use of a slightly intelligent DHCP server that leases IPs for a only a short time frame (on the order of tens of minutes) and only offers IPs that have not been used recently, most networks already using DHCP can quickly change to a randomized scheme. This simplicity does come at a cost, however: TCP connections die whenever the IP change occurs, forcing the hopping period to be quite long or risk unacceptable connection losses.

NASR attempts to reduce this issue by introducing additional intelligence into the DHCP server to allow it to detect "long-lived" TCP connections (i.e., a download) and give clients the same IP if they appear busy. Beyond that, they also monitor what services each client uses, as many are resilient to a connection being torn down [AAMA07]. Despite those improvements, address changes occur even in the fastest of instances only once an hour or so. As tests show, this meets the goal of hitlist worm protection, but is likely inadequate for obfuscating the network from a more intelligent, focused enemy.

2.5.5 Network Address Hopping (NAH).

A 2005 paper by European researchers presents a system they call Network Address Hopping (NAH) [SSH05]. This system focuses on a client contacting a server as a negotiable protection measure, rather than an always-on system used between pre-configured systems.

A protocol employing IPv6 allows a client to tell a server that they supported (and wish to use) NAH. If the server supports NAH, it replies with its hopping pattern. The client sends its own hopping pattern, before reconnecting using the pattern the server just gave it. Hops occur based on packet count per connection, rather than a time-based factor.

Once again, the NAH authors note the importance of encryption in maintaining the confidentiality of the data stream. However, they also state that without encryption the system still provides some benefit, as packets bound for "different" addresses might follow differing network routes due to the different (perceived) destinations. If this occurs, an attacker needs to either compromise a route fairly close to an endpoint to ensure they see all traffic or compromise every possible route and collate the traffic together. Even if they manage to accomplish that, the attacker would still have to collect all traffic passing them in order to reconstruct the full stream (because they are unable to filter for specific IPs to identify the connection they are interested in), which poses a storage and computation problem given enough data [SSH05].

As an additional side effect of the variable routing, the researchers noted that such a system may actually increase the throughput and reliability of a system. If multiple routing paths are used, the network may avoid congestion and allow traffic to flow more smoothly [MHCN96]. While this is not an important aspect of this system, the potential does add support to the employment of address hopping [SSH05].

2.5.6 Transparent Address Obfuscation (TAO).

The 2006 TAO project focuses on protection of the Internet as a whole from hitlist-based worms and is somewhat based on the previous work in NASR [AAMA07]. It features gateways on networks that maintain external-to-internal address mappings for all nodes inside the protected network, with the external addresses changing with a configurable frequency. To maintain existing connections regardless of mapping changes, TAO includes a NAT table.

A disadvantage of this design comes in the form of address space overhead, because each individual box receives a publicly accessible IP and the NAT table claims additional IPs for on-going connections. Testing shows that around 10% more address space is needed for three simulations on large-scale networks. However, TAO has the distinct advantage of only requiring the addition of a single box at the network's edge and no cooperation from remote hosts is needed for it to provide its services.

2.6 Summary

This chapter presents background topics used in this thesis, including IP routing, encryption and authentication, and Time-Based One-Time Passwords. IP hopping and the two possible approaches to it—end point or gateway—are discussed, along with an analysis of each of their benefits and challenges. Finally, previous research in network address space randomization is discussed.

III. Implementation

The research this thesis presents relies on a custom IP address hopping solution. This chapter covers many of the details of this system, from high-level architecture to the network protocol. Section 3.1 covers the requirements this system strives to meet. Section 3.2 gives an architecture overview, while Section 3.3 covers the details of each component in the system. Section 3.4 details the network protocol used by the system to coordinate gateways.

3.1 Requirements

ARG focuses on the needs of military networks. These are essentially the needs of any geographically-diverse organization: locations throughout the world, high availability and reliability requirements, and security over any network through which its data travels [Mal97].

ARG primarily attempts to protect communication outside of military-controlled networks and prevent external entities from probing internally. This protection includes privacy, as ARG (like any network address space randomization tool) helps hide information from adversaries by obfuscating sender and receiver information [SK02]. This means that the implementation described is intended to be employed for all traffic traveling between bases, but is unconcerned with internal base traffic. Internal network defense remains the purview of traditional defenses. Base networks, in this case, include both those on permanent installations and those in deployed, forward locations.

ARG must operate over the commercial Internet. Some proposals for network address space randomization require changes to the Internet's existing routing infrastructure and protocols [WL03, APWJ03]. Deploying such a solution may be possible and beneficial in the long run, but a solution that could be deployed today without participation from outside

entities is more feasible. This is especially true for forward locations, where traffic is more likely to utilize infrastructure outside the military's control.

Like any large, distributed organization, bases can generate huge volumes of traffic, so ARG must scale well. Due to the importance of networks to command and control, ARG's implementation must not introduce significant latency under any foreseeable load. Likewise, there can never be a brief period where all connections drop. At a minimum, given the number of nodes inside the network, dropped connections result in a massive amount of wasted bandwidth as they are reestablished and the data retransmitted.

Military networks contain a wide range of hardware and software. Much of this software cannot be altered to accommodate ARG, so it must function transparently. Host-level implementations might be feasible for generic workstation images (i.e., an alteration to the operating system's network stack), but the ability to function in another way must exist to allow legacy equipment to continue operating.

3.2 Architecture Overview

As illustrated in Figure 3.1, ARG functions entirely around standard networks with hopping gateways. This matches the "gateway hopping" scheme discussed in Section 2.2.2. As with [AA06], these gateways are standalone systems, not intended for use with other tasks. Individual hosts inside these networks have no knowledge of the traffic transformations the gateways perform, whether their connections are routing to a host inside the local network, to a host inside another associated hopping network (hereafter referred to as an "ARG network"), or to an external network. The implementation of ARG allows the deployment of standard passive defense technologies like firewalls inside the network without reconfiguration. Each gateway maintains a NAT-like table to ensure that existing connections are maintained across hops (essentially temporarily leaving the old IP active for just those connections using it already).

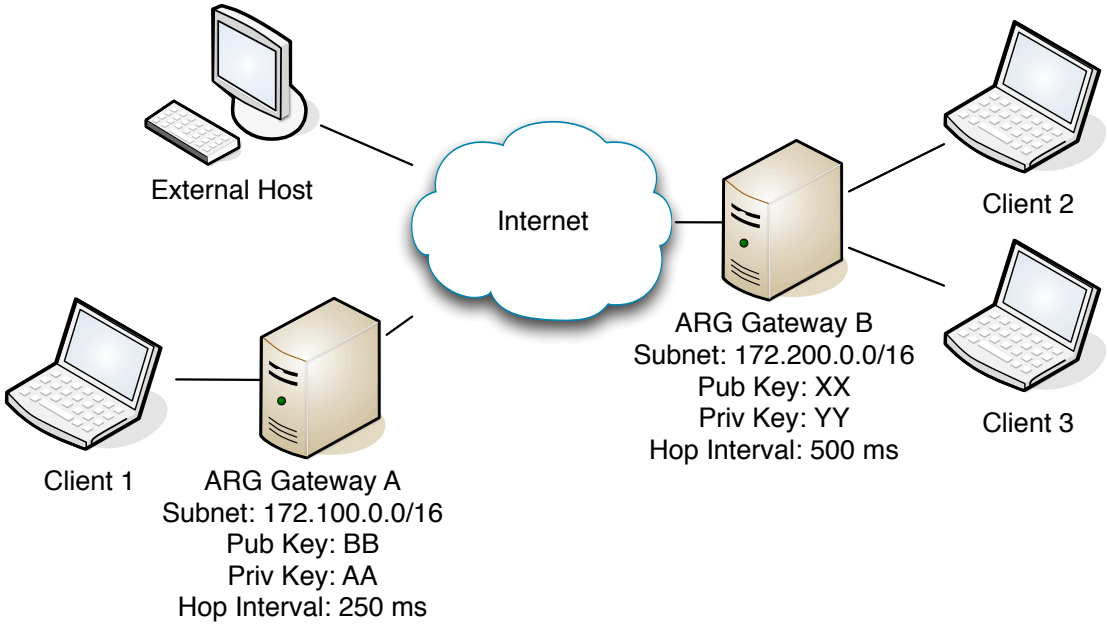


Figure 3.1: ARG conceptual network layout

Each gateway is given what subnet it is permitted to hop within, a private/public key pair, and time it should wait between IP address changes (frequently referred to as its "hop interval"). For example, ARG Gateway A in Figure 3.1 uses subnet 172.100.0.0/16, has the specified public and private keys, and hops every 250 milliseconds. Additionally, each gateway is pre-configured with knowledge of at least one other gateway. The configured information consists solely of the subnet the other gateway is handling and a public key to use for authentication with it. In the figure, this would mean that Gateway A knows that Gateway B sits in the 172.200.0.0/16 subnet and has the public key XX, but nothing else. The gateways transfer additional information during the connection process.

At startup, each gateway generates a random symmetric encryption key and a "hop key." They then attempt to connect to all other gateways for which they have configuration files through a series of data and time synchronization packets. Once two gateways fully connect, they exchange data about all other gateways they have configured, allowing ARG

nodes to connect to gateways for which they do not have physical configuration files. Periodically gateways re-send time synchronization information, ensuring that changing network conditions do not kill communication. Section 3.4 and Appendix B cover each of these packet exchanges in more detail.

Once connected, packets between ARG networks are encapsulated, encrypted, and authenticated by the originating network's gateway, with each packet given the destination network's current IP address. On receipt, a gateway checks that the IPs match what they expect (both its own IP and the source's IP), then validates the authentication information (signature/HMAC) before forwarding the original packet into their network. Packets to external hosts flow through the NAT-style system covered in Section 3.3.3.

During the evaluation of IPs, gateways allow packets to match either the current address or the previous one. This allows packets sent just before an IP change to still be accepted. If a gateway's hop interval is shorter than twice the one-way latency, packets will never be accepted, as illustrated in Figure 3.2. In this figure, Gate A sends a packet to Gate B with the destination addresses set to the current IP at that time (172.200.210.85). The packet takes 50 milliseconds to travel across the network, during which time the gateways change addresses twice. When the packet reaches Gate B it is rejected, because the packet's destination IP does not match the current (172.200.38.138) or previous (172.200.60.97) IPs.

3.3 Components

The handling of packets within ARG is distinctly different if they are to an external (non-ARG network) host or to an ARG network. These processes are handled by two separate components, the hopper and the NAT. A high-level director decides which of these two receives each incoming packets. All of these components run as separate threads on the same gateway, closely coordinating their work. Because it is in charge of overall system operation, this section begins by discussing the director.

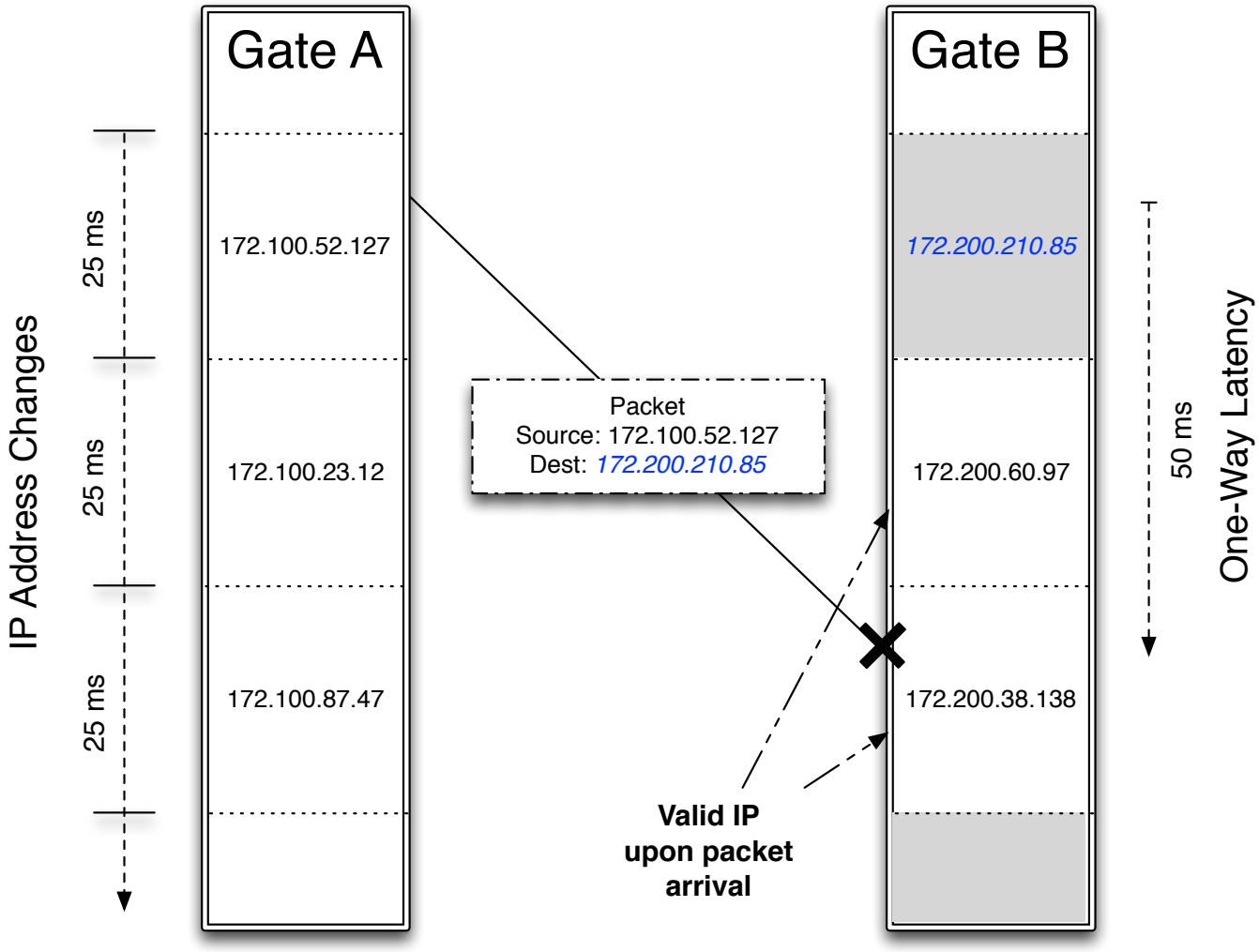


Figure 3.2: Packet sent between gateways when hop interval is half the latency

3.3.1 Director.

The director is in charge of receiving packets on the internal and external interfaces of the gateway. Upon receipt of a packet, the director parses the packet and decides how to handle it. The director's decision tree is illustrated in Figure 3.3 and discussed below.

In the case of ARP requests on either interface, the director replies with the gateway's MAC address. This feature of ARG allows its use without any changes to the network it is placed in, as hosts continue to send to the "same" gateway IP as before and ARG responds,

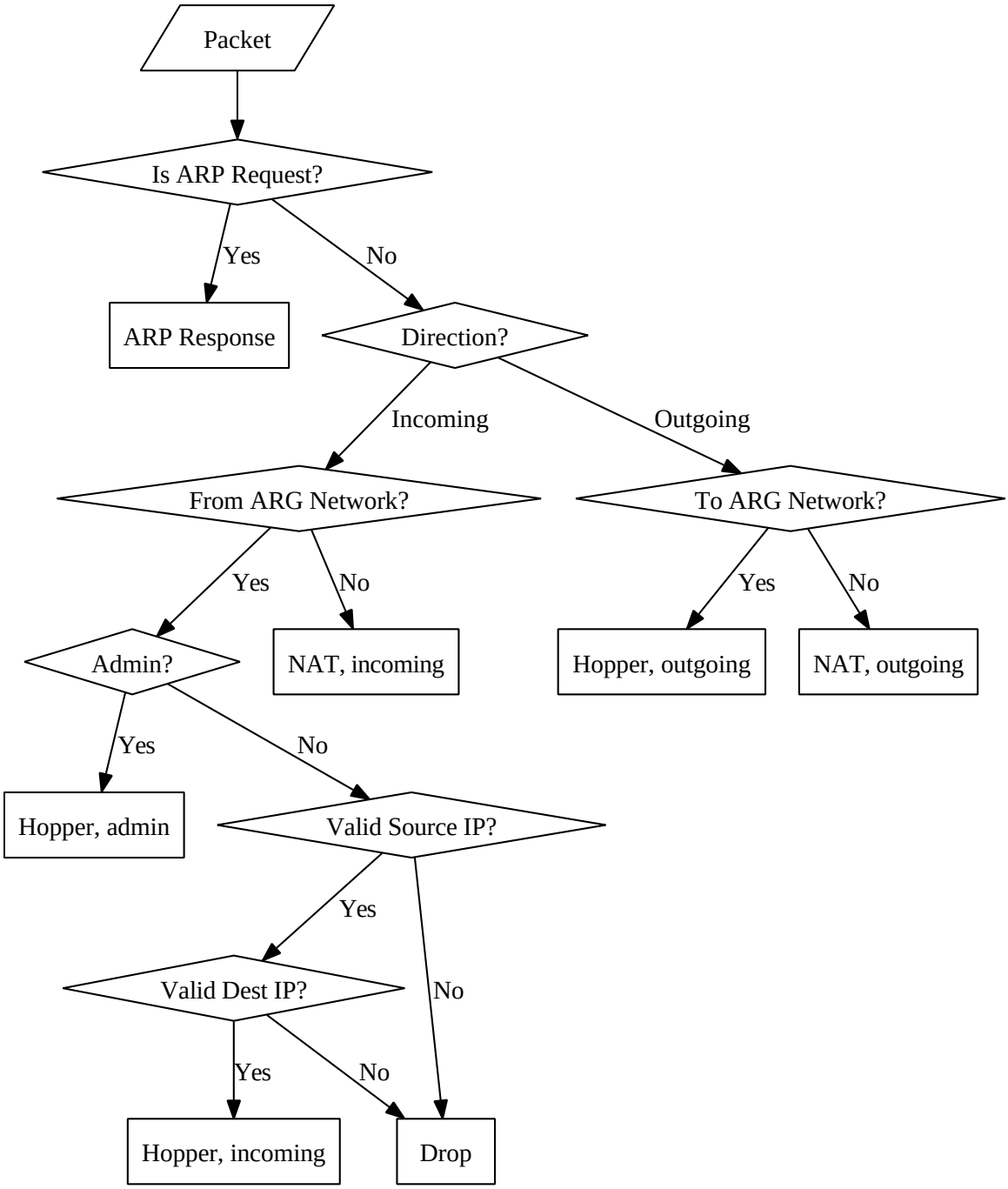


Figure 3.3: ARG director flow

despite not technically possessing an internal IP of its own. For example, the inside hosts send out an ARP request for their normal IP gateway (as Section 2.1.3 discusses) and

the ARG gateway sends an ARP response, allowing it to grab all traffic and process it appropriately. More details on Ethernet and IP routing can be found in Sections 2.1.3 and 2.1.1.

For outgoing packets (packets that are sent from within the protected network that are intended to leave the network), the director checks the destination IP of the packet. If the IP is unknown, it is passed to the NAT module. If the IP is within another ARG network the director knows, the packet is handed off to the hopper module to be wrapped and transmitted.

For incoming packets (packets hitting the external interface), the director first checks if the source IP is another ARG network. If it is *not*, the director quickly hands the packet off to the NAT inbound handler.

If the packet *is* from another ARG network, the director checks if the packet type indicates it is an administrative packet and hands it off to the hopper's administrative processor. If it is not an administrative packet, the director confirms the gateway which sent the packet is actually connected. Which gateway is determined by matching the source IP to the base IP and mask in the local gateway's configuration. Assuming that the gateway is connected, the local gateway checks that the source and destination IPs are correct, based on the data the hopper has on the other gateway. Assuming all checks pass, the packet is handed off to the hopper to be unwrapped and forwarded into the network. If any fail, the packet is silently dropped.

3.3.2 Hopper.

The hopping module is the heart of ARG. It maintains the state of the gateways (e.g., keys, hop intervals, times) it knows about and transfers packets to and from the network it is protecting. The other two components (director and NAT) talk to the hopper to obtain current IP information and if a given gateway is connected or not.

When the hopper first starts, it initializes a list of gateways with data from configuration files. The information maintained in this structure is shown in Table 3.1.

Table 3.1: Information hopper module maintains on other ARG gateways

Data	**Information Source**
IP Range (IP and mask)	Configuration file (see Appendix D)
RSA Public Key	Configuration file
Hop Interval	Transferred during connection (see Section 3.4)
Symmetric Key	Transferred during connection
Hop Key	Transferred during connection
Time Base	Calculated based on latency (see Appendix B)

An administrative thread is started at the same time. This thread attempts to connect to each gateway in its list periodically and, if it does not hear from a given gateway for several minutes, marks gateways as disconnected. In addition, it sends periodic time synchronization requests to connected gateways, especially if it sees a large percentage of packets being rejected due to incorrect IP addresses.

Beyond the administrative thread, actions occur in the hopper only when the director passes packets off to it. Outgoing packets are always wrapped, encrypted, and signed, as covered under the "route packet" process in Section B.2.4. Incoming packets go through the validation process shown in Figure 3.4 before being handled. Note that IP checking is done in the director before control reaches the hopper. After validation exact handling depends on the packet type, but is generally covered in Section 3.4 as part of the protocol discussion.

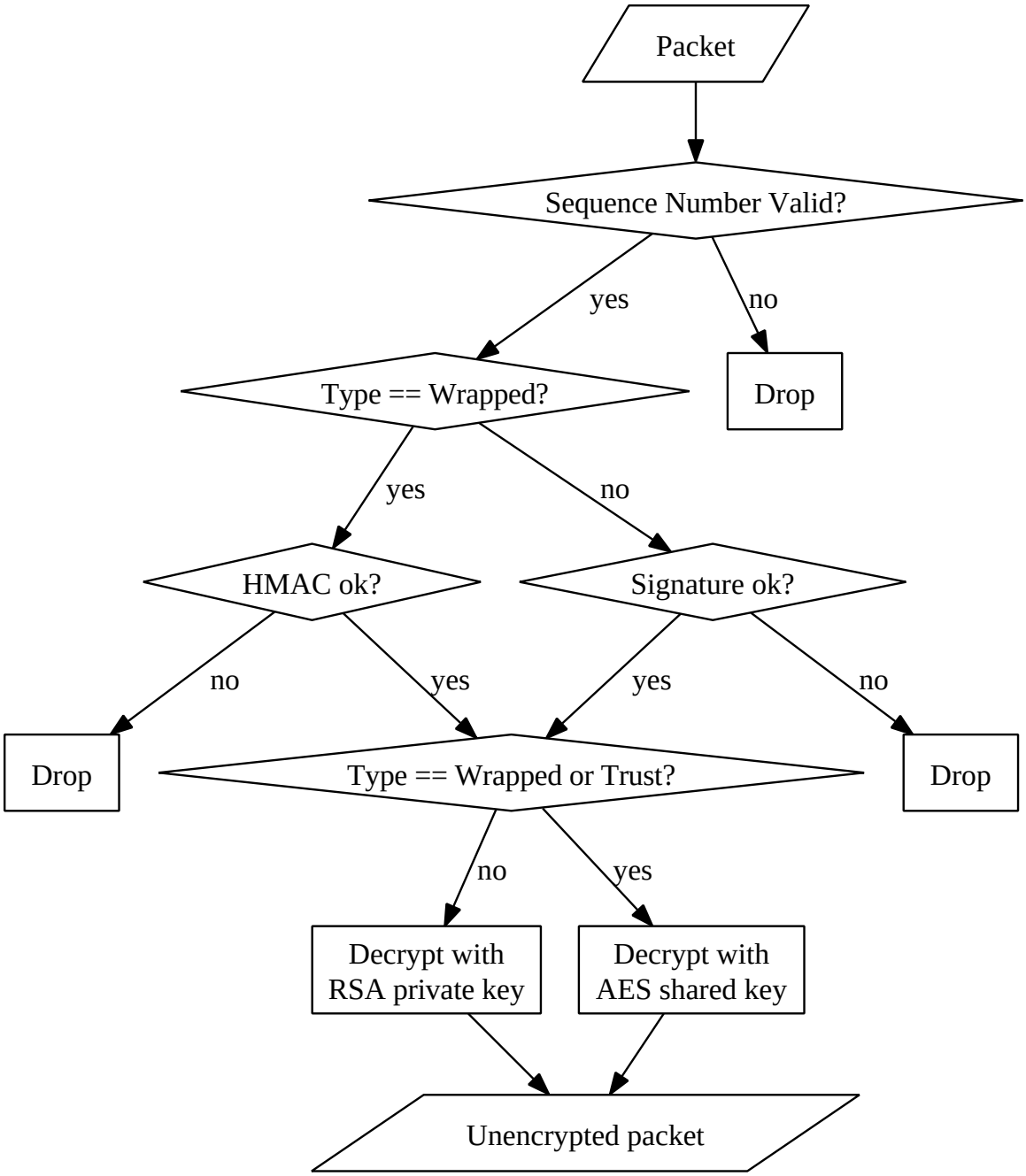


Figure 3.4: ARG incoming packet validation process

3.3.3 Network Address Translator.

The NAT component of ARG maintains a list of on-going connections to external hosts. For instance, if a client inside a protected network connects to 72.246.189.120, the

NAT creates an entry in an internal table and allows packets from the external host back in to the network and to the client. This is almost identical to the operation of the basic NAT discussed in Section 2.1.2. The only difference from a normal NAT system is the addition of an extra field in the NAT table for the IP at the time the connection was first established. The new version of the table with example data is shown in Table 3.2 with the new column in italics.

Table 3.2: ARG NAT table example

	Int IP	**Int Port**	**Remote IP**	**Remote Port**	***Ext IP***	**Ext Port**
1	192.168.0.103	3547	74.125.225.69	443	*172.1.123.35*	50003
2	192.168.0.103	8751	207.109.73.34	80	*172.1.73.1*	42630
3	192.168.0.112	30452	4.27.2.253	80	*172.1.86.173*	53920

The traditional NAT processing and logic is supplemented with this additional information. When a packet goes out, the table is checked and the packet has its source IP address and port changed to the external values. If this is the first packet in a connection, the external IP is filled from the current IP of the gateway. When an incoming packet is encountered, the external IP and port are both checked to determine the correct internal host. The addition of the external IP to the table allows connections to survive across hops; many connections last longer than ARG's intended hop rate and severing connections frequently is unacceptable for many applications.

3.4 ARG Protocol

The ARG protocol is designed to be fairly stateless, simplifying the implementation and lowering the likelihood of an exploit forcing the gateway into an unexpected state. ARG sends data between gateways as the direct payload of the IP packet; transport-layer

protocols such as User Datagram Protocol (UDP) and TCP are not used. If needed, the protocol could be adapted easily to work as a UDP payload. ARG packets are identified by IP protocol 253, a protocol reserved for experimentation [Nar04]. All packets sent between ARG gateways use the header structure shown in Table 3.3.

Table 3.3: ARG packet data

Data	**Size**	**Data Type**
Version	1 byte	Unsigned Integer
Message Type	1 byte	Enumeration, see Table 3.4
Message Length	2 bytes	Unsigned Integer
Sequence Number	4 bytes	Unsigned Integer
Signature	128 bytes	Raw
Payload	0-32,629 bytes	Message-type specific, see Appendix B

For this research, the protocol version field is set to 1 at all times. The type field tells the receiving gateway how to process the data contained in the message. Possible values are shown in Table 3.4. More details on the format of each message type are given in Appendix B. Length is the network-order size in bytes of the data from the version to the end of the data; given the size of the message header, the minimum for this is 136. The sequence number is a monotonically increasing unsigned integer value used to prevent replay attacks.

The signature field may actually contain two possible values: a true RSA digital signature of the packet or an HMAC of the packet, depending on the message type. Packets of type `PING` or `CONN_REQ`/`CONN_RESP` are encrypted with the public key of the receiver and signed with the private key of the sender. All other packets—types `WRAPPED` and `TRUST_DATA`—are encrypted with the symmetric key of the receiver and include an HMAC

Table 3.4: ARG message types

Mnemonic	Value	Description
WRAPPED	1	Encapsulated packet from protected client to protected client
PING	2	Time synchronization message
CONN_RESP	3	Connection data response message
CONN_REQ	4	Connection data and request for other gateway's data
TRUST_DATA	5	Configuration information

of the encrypted data using with the symmetric key of the sender, following standard encrypt-then-MAC practice [BN00]. The encryption and signing combinations used for each message type as well as whether or not the source and destination IP addresses are strictly checked are summarized in Table 3.5.

As a side note, the encrypt-then-sign order used by ARG (when using a digital signature rather than an HMAC) is backwards according to best-practice RSA and should be corrected to sign-then-encrypt [AN95]. It may also be wise to add the name of the sending gateway to each message to help prevent similar mistakes in the future [AN94]. The order ARG uses allows an attacker to sign a message and claim it as their own [Ram99]. However, ARG gateways will still reject these falsified messages because the receiving gateway will not have a public key for the signer. A gateway could successfully sign another gateway's message, but this indicates a compromised of the gateway itself, at which point the attacker has full access to the entire network anyway.

There are four basic exchanges that happen between ARG gateways: connect, time synchronization, trust data exchange, and packet transfer. In order for gateways to begin exchanging packets between the networks they are protecting (via the "route packet" process), they must first fully connect by completing the connect and time synchronization

Table 3.5: ARG message security summary

Type	Encryption	Signing	IPs checked?
WRAPPED	AES, remote key	HMAC, local key	Yes
PING	RSA, remote public key	Signature, local private key	No
CONN_RESP	RSA, remote public key	Signature, local private key	No
CONN_REQ	RSA, remote public key	Signature, local private key	No
TRUST_DATA	AES, remote key	HMAC, local key	No

processes. The trust data exchange step is optional, although it allows gateways to connect to others without configuration files. The precise requests, receives, and verifications for each exchange are given in Appendix B.

3.5 Summary

This chapter discusses the implementation of ARG. It begins with the requirements ARG fulfills, then covers the high-level architecture of the system. The chapter then examines each component and the protocol ARG uses between gateways.

IV. Methodology

This chapter discusses the methodology used to measure the effectiveness of ARG at correctly classifying valid and invalid traffic, the shortest supportable hop interval at various network latencies, the maximum packet rate ARG can handle, and the overall stability of the system under test. Section 4.1 discusses the problem this research seeks to answer. Section 4.2 defines the System Under Test (SUT), and Section 4.3 goes into detail on the possible outcomes of the Component Under Test (CUT). Section 4.4 covers the workload presented to the SUT, Section 4.5 covers the configurable parameters of the SUT, and Section 4.7 covers the metrics collected. Sections 4.6 and 4.8 detail the actual tests and the purpose of each.

4.1 Problem Definition

4.1.1 Goals and Hypothesis.

This research tests whether network address space randomization as discussed in Chapter 3 is suitable for deployment on a corporate or military network. Tests against this system are designed to answer four basic questions:

1. Does ARG classify traffic correctly? What percentage of false positives (valid packets blocked) and false negatives (invalid traffic allowed through) does it introduce?

2. What is the maximum packet rate and throughput ARG can support?

3. What is the minimum supportable time between hops? How does latency affect this?

4. Is ARG stable when presented with corrupt, malformed, or replayed packets?

It is hypothesized that ARG correctly classifies 99% of traffic it encounters when operating with a hop interval appropriate for the network latency. In addition, this thesis

hypothesizes that packet loss becomes acceptable when the hop interval matches or exceeds the one-way network latency, where acceptable loss is defined as less than 2%. This percentage is based on the loss seen on Massachusetts Institute of Technology's wireless networks [Tal12]. The other two questions are informational as the results apply only to this specific hopping gateway implementation, but it is believed that ARG is stable in the face of malformed traffic and it can handle at least 10 megabits per second (Mbps) of traffic.

4.1.2 Approach.

This research is accomplished on a test network with nodes representing the types of hosts found on a typical, corporate-style network. These include trusted hosts inside trusted networks which communicate freely, internal and external servers that must be accessible to hosts inside these trusted networks, and malicious hosts outside the networks. A configurable custom hopping gateway sits in front of the trusted networks.

Traffic generators and collectors run on the test network, determining which traffic flows successfully make it to their intended destination. This includes examining both false positive and false negative rates, determining why ARG rejects packets that should get through and why it allows packets that should be rejected. After a given test, logs and traffic captures are collated to form a complete picture of the traffic on the network before determining statistics.

4.2 System Boundaries

The SUT is ARG, the custom IP hopping gateway developed specifically for this effort. The basic components of this system, the various inputs into the system, possible outputs, and the metrics provided are illustrated in Figure 4.1. The sections following cover aspects of this diagram in more detail, with Section 4.3 discussing the possible outcomes, Section 4.4 covering the workload, Section 4.5 detailing the parameters in use, and Section 4.7 covering the metrics collected.

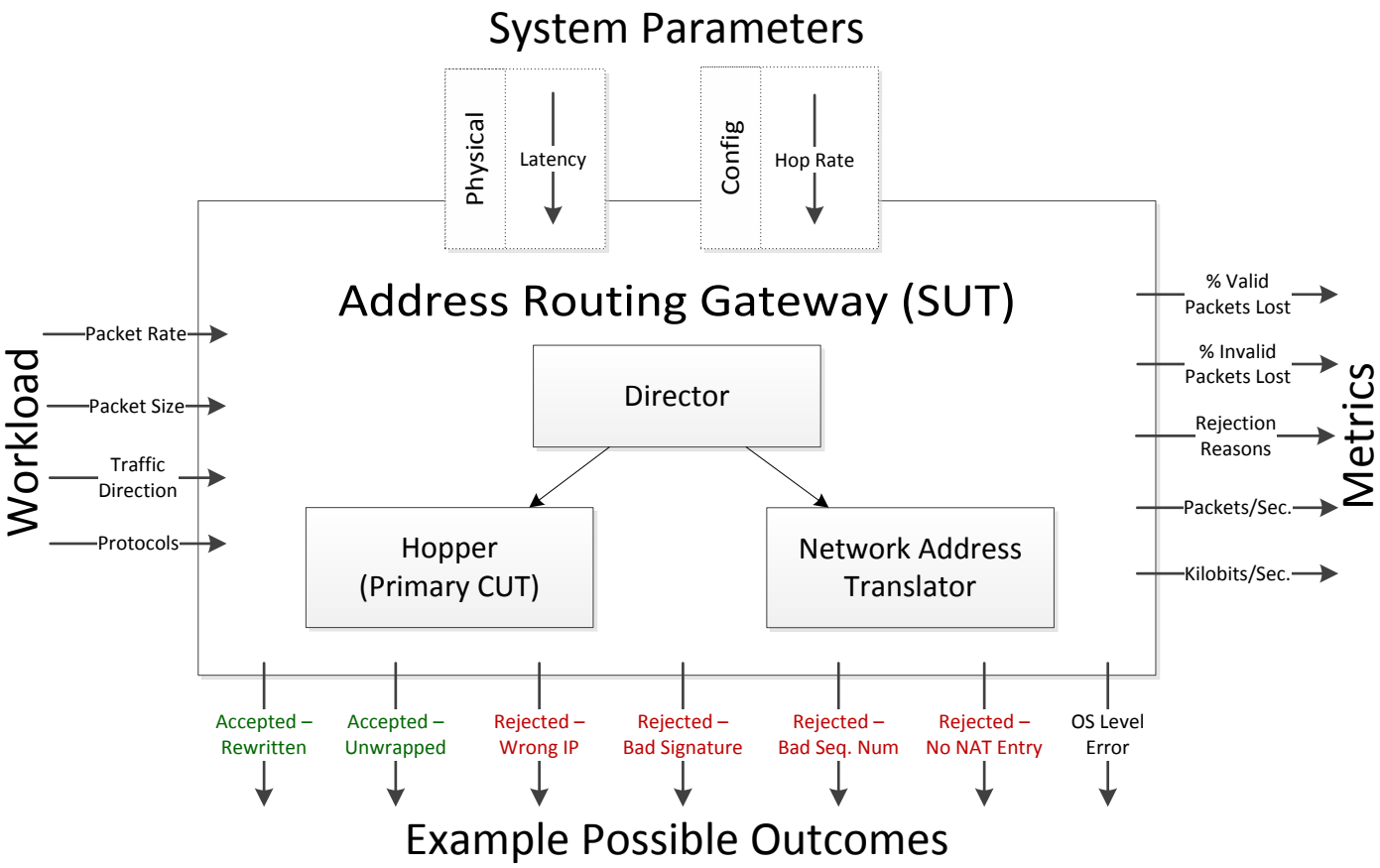


Figure 4.1: ARG SUT diagram

4.3 System Services

This thesis tests three components of ARG. Most important is the hopper module, which provides a rapidly-changing external IP address and details on connected ARG networks. Using this information, it transports packets between ARG-protected networks. Packets to and from external hosts—hosts that are not part of an ARG network—go through the NAT module. Finally, the director module hands packets off to each of the other modules and collects the results back to be logged and potentially acted upon. More details on these components are in Chapter 3.

The potential outcomes of the director are shown below, broken into separate sections based on incoming or outgoing packets. Other services do not directly offer outcomes relevant to this research.

- Director - Incoming
  - Accepted: Rewritten and forwarded - Packet is from non-ARG network and is rewritten via NAT table before forwarding.
  - Accepted: Unwrapped and forwarded - Packet is from ARG network and passes validation checks. Contents are extracted and forwarded internally.
  - Rejected: Incorrect source IP - Packet is coming from an ARG network but does not have what the local gateway believes is the current source IP for the other gateway.
  - Rejected: Incorrect destination IP - Packet is coming from an ARG network but does not have the current local gateway IP as the destination.
  - Rejected: Incorrect message size - The message length does not match the message type.
  - Rejected: Incorrect sequence number - The message's sequence number is not monotonically increasing.
  - Rejected: Unable to verify signature/HMAC - Packet signature invalid/nonexistent (if coming from an ARG network).
  - Rejected: No NAT bucket/entry - Packet is coming from a non-ARG network but does not have a valid entry in the NAT table.
  - Rejected: Misc - Some operating system-level errors may occur, resulting in rare errors in sending or receiving packets.

- Director - Outgoing
  - Accepted: Rewritten and forwarded - Packet is destined for non-ARG network. An entry is made/retrieved from the NAT table and used to rewrite packet.
  - Rejected: Gateway not connected - Packet was intended for an ARG network the gateway is aware of but not yet connected to.
  - Rejected: Wrapped and forwarded - Packet is destined for an ARG network. Wrapped and placed on the external network.
  - Rejected: Misc - Some operating system-level errors may occur, resulting in rare errors in sending or receiving packets.

4.4 Workload

Workload to the system is the traffic flowing through the ARG gateways. Standard network traffic parameters like packet rate, packet size, protocol types, number of simultaneous ongoing connections, and lifetime of connections play a role. For the purposes of this thesis, however, packet rate is the primary factor. Each test involves traffic generators, all of which may be instructed to wait a given amount of time between each sent packet. The lower the packet delay, the higher the packet rate.

It is important to note that network performance itself is not a large concern of this research. Packet rate does provide useful information about the performance of ARG, but the numbers apply only to this specific implementation. ARG's development does not focus on performance in this first iteration, leaving many possible areas for improvement. Previous research has shown that similar solutions have minimal impact on performance [SSH05].

All traffic generators in the tests create randomly-sized packets of either UDP or TCP traffic. The protocol used depends on the test being run at the time; Section 4.8 discusses each test series and the traffic flows they utilize.

4.5 System Parameters

As a network application, ARG is affected by both the machine on which it runs and the network over which it communicates. ARG's local performance is most affected by processor and memory speeds, with encryption potentially consuming a fair amount of processor time and memory speeds impacting virtually all aspects of operation.

The primary physical network parameter that affects ARG is latency. To ensure that two ARG gateways are able to communicate reliably, packets sent from one gateway to the other must arrive before the IP addresses used in the send are no longer current. If hops occur too frequently, a high one-way latency will cause sent packets to frequently arrive after the receiving gateway has hopped to a different IP address. (Adapting to latency is an area of potential improvement, as Section 6.3 discusses.) The test environment inherently introduces less than one millisecond of latency, but artificial latency can be added to simulate a more realistic range of network conditions.

The primary configuration setting for ARG is the hop interval. ARG allows the time between hops to be customized from several times a second to minutes apart with millisecond precision. Each gateway may be configured to hop at different rates, but for the sake of this thesis the hop intervals for each gateway are identical in a given test.

4.6 Evaluation Technique

Measurement is used to obtain results for each factor level. Due to the fairly complex interactions needed between ARG gateways and the processing needed to decide how to handle packets, simulating the system would likely require an equal amount of work with little benefit.

Setup of the test environment involves a basic seven-node network: three gateways running ARG, one system on the network protected by each gateway, and one host outside the network. Figure 4.2 shows the network and the names given to the various systems.

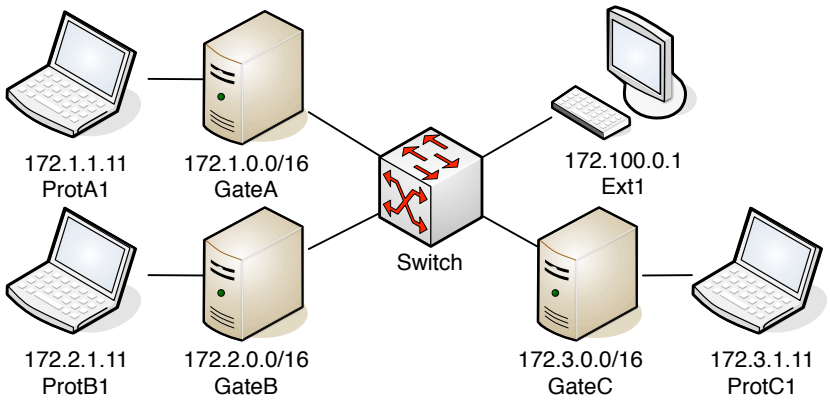


Figure 4.2: ARG test network layout overview

Protected clients behind the gateways (`ProtA1`, `ProtB1`, and `ProtC1`) may communicate freely. The protected clients may also talk out to the external host (`Ext1`), and the external hosts must then—once that connection is established—be able to talk back into the network. There is additional administrative traffic directly between the gateways (`GateA`, `GateB`, and `GateC`). These three basic traffic flows are "valid" traffic.

All traffic beyond what is described above is "invalid." For example, `Ext1` is not allowed to send traffic in to the protected clients or the gateways without them first initiating the connection. Malformed traffic sent by any host is also considered invalid. In either case, invalid traffic should be stopped at the earliest possible opportunity (i.e., the gateway rejects the packet and keeps it from reaching the internal host) and the gateway must remain stable.

To collect data, each system runs the traffic collection program `tcpdump` to capture traffic sent and received into Packet Capture (PCAP) files. The test execution script then spawns traffic generators on the correct systems in the network, based on what test is being run. Section 4.8 details the types of traffic each test establishes. Each traffic generator logs their sends and receives to text log files (one per generator), independent of the `tcpdump` traffic captures. After a given trial, the PCAP files and traffic generator logs are collated and processed with custom scripts to determine the metrics described in Section 4.7. More

details on the traffic generators and test run sequence can be found in Appendixes C and E. Appendix F covers the custom results processor.

All trials run on a network of seven physical servers. Each server runs Ubuntu 12.04.1 Server Edition with four gigabytes of Random Access Memory (RAM) and a 2.6 gigahertz quad-core Intel Xeon. A single switch with four Virtual Local Area Networks (VLANs) connects each system at 100 Mbps.

4.7 Performance Metrics

As previously stated, this research primarily focuses on the classification accuracy of ARG and the interaction of the hop interval and latency. Measurements on ARG therefore concentrate on the outcomes from the director. However, basic statistics on network performance are collected. The metrics of interest include:

- Percentage of invalid packets accepted

  If a packet that should have been rejected is accepted by ARG, it is possible for an attacker to sneak into the network regardless of the gateway's existence. This is the true measure of whether or not ARG is protecting the network. If ARG functions correctly, this number should remain at zero for all experiments with ARG enabled.

- Percentage of valid packets rejected

  In ideal circumstances, this will also be zero. However, network conditions may result in failures here, which on a real-world network might result in a disruption of service.

- Number of each type of rejection (each possible outcome from the director)

  This reveals where in the processing stage packets are typically caught. If packets get caught in the later stages of validation—e.g., signature checking—then processing time has been wasted.

- Packets per second and kilobits per second (Kbps)

  An easy check on ARG's performance is comparing the amount of traffic it is handling against the packet loss it shows. Information about both the number of packets and the raw number of bits it processes may reveal slightly different results, so both are collected.

4.8 Experimental Design

Based on the system and workload parameters given in Sections 4.4 and 4.5 and the goals of this research (as presented in Section 4.1.1), there are eight traffic flows of interest. Each consists of different types of traffic and flow destinations. These are most easily visualized in Figure 4.3 on page 49.

These possible traffic flows are used in four sets of experiments, given below. Each experiment set answers a different research goal and utilizes different factor levels, as shown in their respective tables.

- Basic tests

  Table 4.1 displays the factor levels used for this series of tests. This sequence of tests verifies that ARG classifies traffic correctly by running every test shown in Figure 4.3 against ARG. Latency is set to 20 milliseconds (Round-Trip Time (RTT)) for all tests and traffic generators produce packets around every 0.3 seconds. To determine if the time between hops has a statistically significant impact on certain types of traffic, every test runs twice, once with a long hop interval of 500 milliseconds and once with a shorter interval of 50 milliseconds, which is just slightly more than double the round-trip latency.

- Maximum Throughput

  Table 4.2 displays the factor levels used for this series of tests. This sequence gives an indication of what throughput and packet rate ARG is capable of handling. Packet

Table 4.1: Factor levels for basic tests

Factor	**Possible Levels**
Hop interval (ms)	500, 50
Round-trip latency (ms)	20
Packet delay (s)	0.3
Traffic direction and type	Flows 0–8 (See Figure 4.3)

delay goes through all levels shown, which leads to roughly corresponding increases in the throughput the gateways must handle. As with the basic tests, the hop interval alternates between 500 ms and 50 ms to see if the additional IP calculation load impacts the maximum rate. Flow 4 is used across all runs to because it utilizes both TCP and UDP traffic flowing in all valid directions. RTT is set to 20 ms.

Table 4.2: Factor levels for throughput tests

Factor	**Possible Levels**
Hop interval (ms)	500, 50
Round-trip latency (ms)	20
Packet delay (s)	0.2, 0.1, 0.05, 0.01, 0.005, 0.001
Traffic direction and type	Flow 4 (See Figure 4.3)

- Minimum hop interval

  Table 4.3 displays the factor levels used for this series of tests. This sequence determines the minimum time between IP address changes at various latencies. The hop interval and latency go through the levels shown in a full factorial fashion (every

latency-hop interval combination). Packet rate is fixed at 0.3 seconds. Flow 4 is used throughout.

Table 4.3: Factor levels for minimum hop interval tests

Factor	**Possible Levels**
Hop interval (ms)	1000, 500, 300, 200, 100, 75, 60, 50, 40, 30, 15, 10, 5
Round-trip latency (ms)	0, 30, 100, 500
Packet delay (s)	0.3
Traffic direction and type	Flow 4 (See Figure 4.3)

- Fuzzer

  Table 4.4 displays the factor levels used for this series of tests. This sequence is not tested rigorously for traffic flow success and failure, but ensures that ARG remains stable despite malformed traffic. Traffic Flow 8 is used, with additional traffic coming from fuzzers running that replay and/or alter all gateway traffic they see. Hop intervals vary between 500 ms and 50 ms, latency is fixed at 20 ms, and packets are sent at 0.3 second intervals.

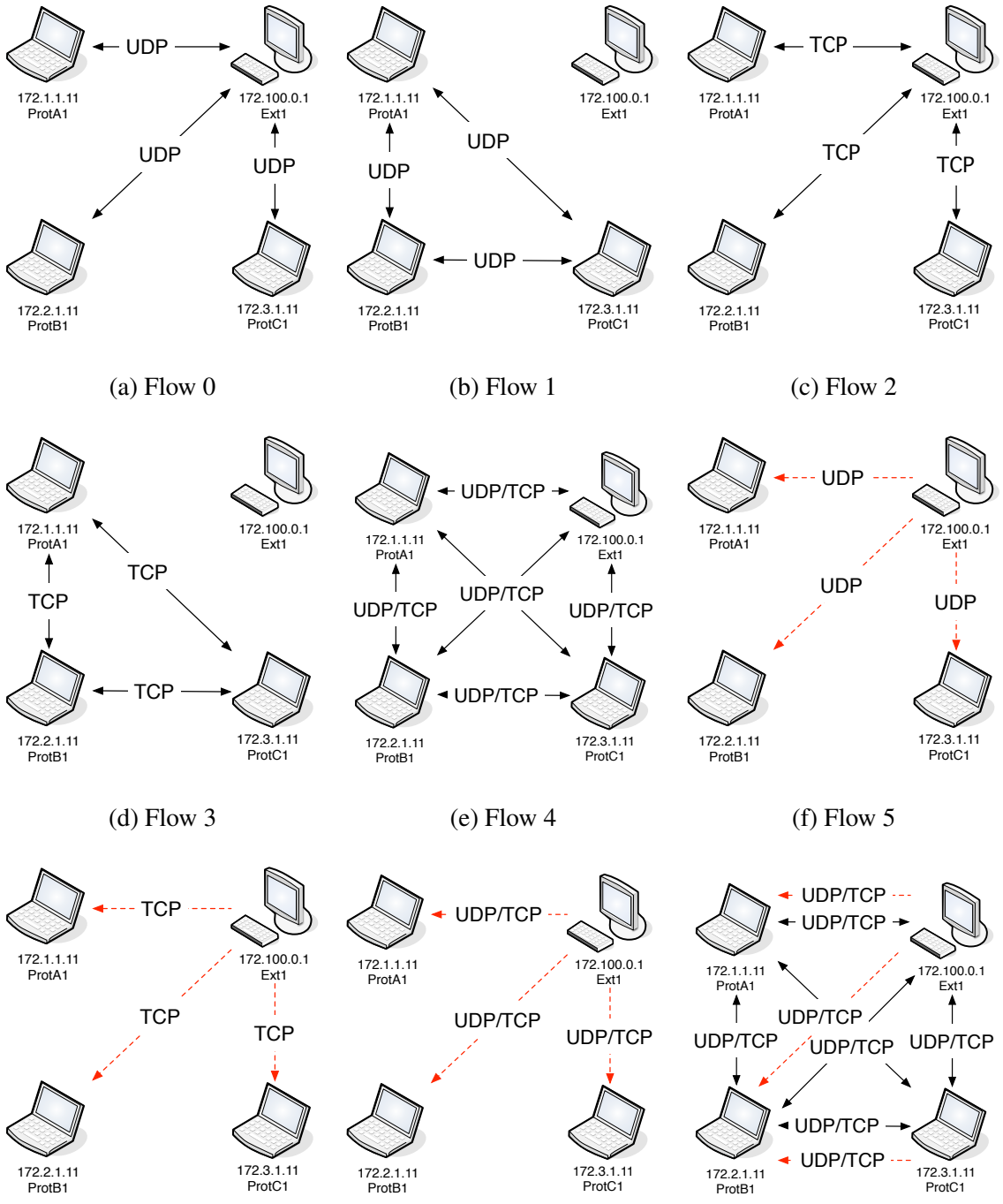
Table 4.4: Factor levels for fuzz tests

Factor	**Possible Levels**
Hop interval (ms)	500, 50
Round-trip latency (ms)	20
Packet delay (s)	0.3
Traffic direction and type	Flow 8 (See Figure 4.3)

A 95% confidence interval is used for all experiments. Experiments are each run for five minutes, sufficient time for the system to stabilize (pilot studies show that ARG fully connects in under 10 seconds on the test network). Some variation is possible in the actual traffic seen in a single run, so a minimum of 10 replications are used for each experiment.

4.9 Summary

This chapter discusses the goals of this research and defines the SUT and its relevant factors. The methodology in use is covered, with details on the test network and the exact tests run on this network. Finally, this chapter enumerates the metrics the tests collect and analyze.



(a) Flow 0 (b) Flow 1 (c) Flow 2

(d) Flow 3 (e) Flow 4 (f) Flow 5

(g) Flow 6 (h) Flow 7 (i) Flow 8

Figure 4.3: Experiment traffic flow directions and protocols. Black solid lines indicate valid traffic, red dashed lines are invalid.

V. Results and Analysis

This chapter presents and analyzes the experimental results. Section 5.1 covers the basic functionality tests, Section 5.2 determines the effects of hop interval and latency, and Section 5.3 discusses ARG's performance. Section 5.4 covers the results of running a fuzzer against ARG. Finally, Section 5.5 revisits the research questions Chapter 4 poses and summarizes the results with respect to each of them.

5.1 Basic Tests

The first set of experiments test the basic functionality of ARG, as Section 4.8 discusses. These tests are intended to verify the basic functionality of ARG and determine if it classifies traffic correctly, per the expected traffic flow. Given that stressing the system is not a goal of this series of tests, packet delay is set to a high value of 0.3 seconds, resulting in a slow packet rate. (Packet rate stressing is left until the throughput tests in Section 5.3.) Hop intervals vary between 50 milliseconds and 500 milliseconds, with a RTT fixed at 20 milliseconds.

5.1.1 Valid Packet Loss.

The raw results for the loss of valid packets on each test are shown in Figure 5.1. Figure 5.2 shows the mean and confidence intervals (CIs) of each test, using a 95% confidence level. Due to a lack of normality in the experimental results, these numbers are calculated via bootstrapping with 1000 replicates.

These figures reveal low packet loss across all test scenarios. At worst, Flow 3—TCP traffic between ARG protected clients only—lost between 0.0588% and 0.7544% of packets, with 95% confidence. Across all tests, valid packet loss averaged 0.1123%, with a CI of $(0.0659\%, 0.1478\%)$, again with 95% confidence. These results confirm Chapter 4's hypothesis that ARG causes packet loss less than 2%, at least under normal conditions.

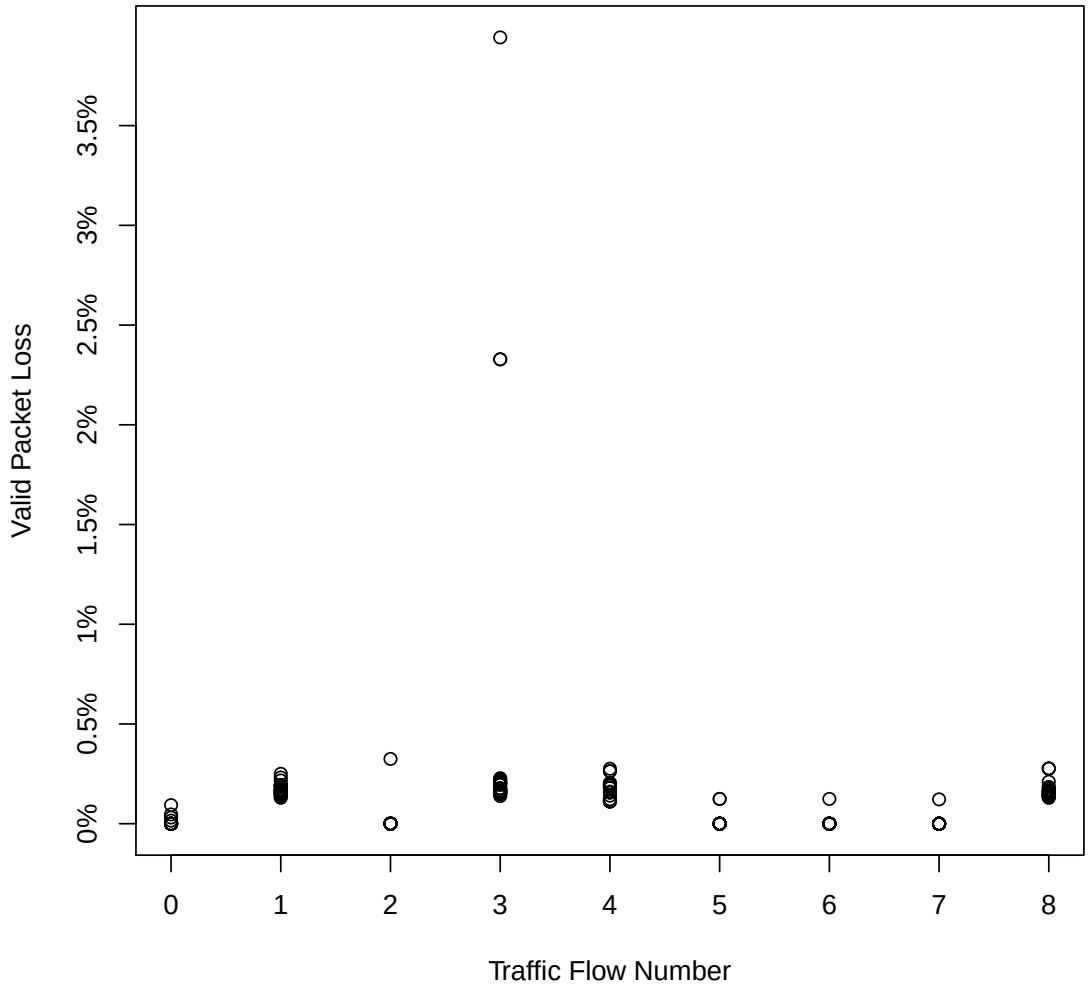


Figure 5.1: Basic tests, raw valid packet loss. Traffic flow numbers correspond to the flows in Figure 4.3.

Figure 5.1 displays two outliers on Flow 3. Examining the data reveals the dropped packets exceeded the maximum size of a packet ARG can pass between gateways. As Chapter 3 covers, ARG wraps packets between gateways in its own headers, increasing the size of the packet. Ethernet II has a default maximum transmission unit of 1500 bytes, so

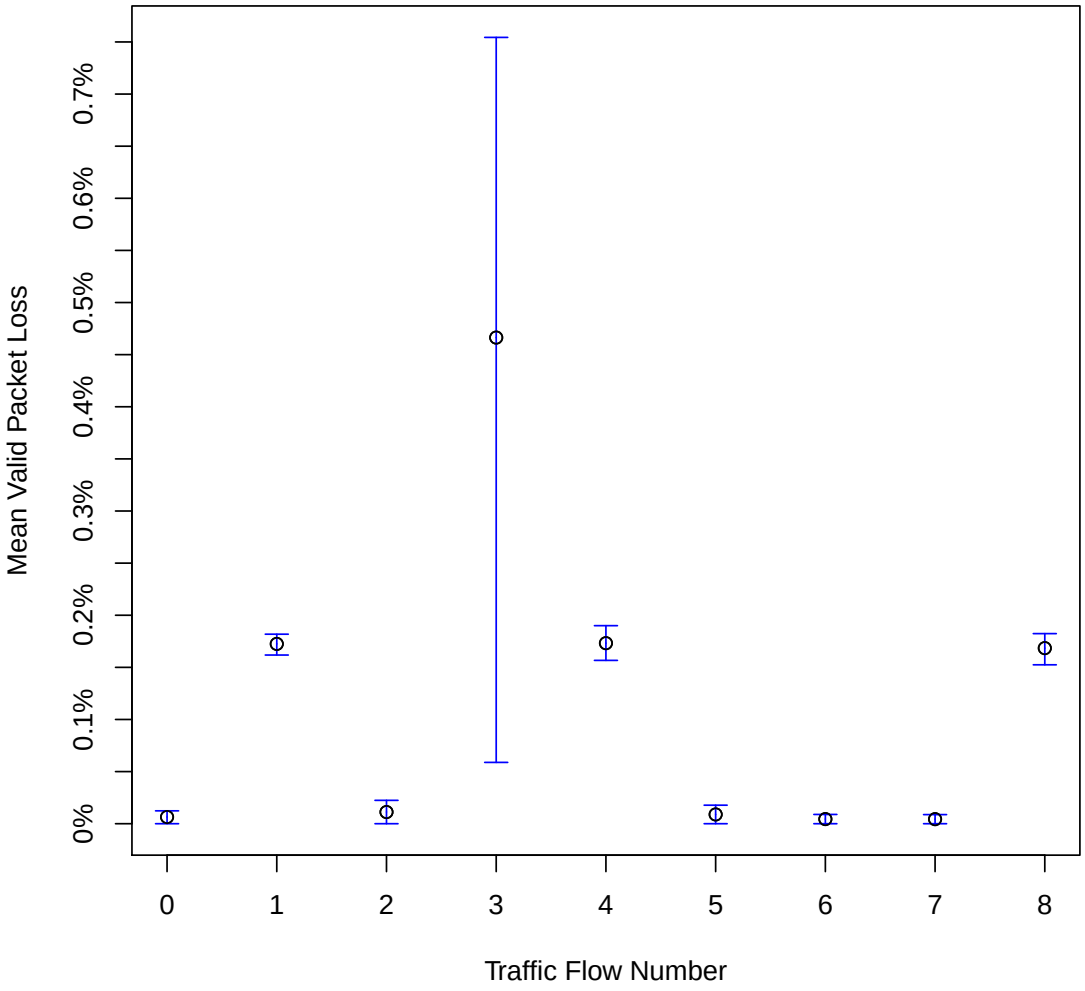


Figure 5.2: Basic tests, valid packet loss means and confidence intervals. Traffic flow numbers correspond to the flows in Figure 4.3

ARG drops packets that pass this limit after being wrapped (ARG is unable to fragment packets or inform the sender that fragmentation is needed, as Section 6.3 mentions). In fact, the maximum packet size issue is the most common reason for packet rejection in the basic set of tests (in tests with only valid traffic), as shown in Table 5.1. While test

traffic is controlled for size, an oversight in the test traffic generators fails to account for TCP options, which can increase the size of the packet beyond the limit. TCP options here refers to the optional header data after the main TCP headers [Pos81].

Table 5.1: Basic tests 1-4 packet rejection reasons. 143 tests with 2,312,228 total packets represented.

Reason	**Count**	**% of Total Packets**
Outbound message too long	949	0.04104%
Outbound sequence number incorrect	596	0.02578%
Inbound unwrapped	578	0.025%
Unknown	518	0.0224%
Outbound gateway not connected	393	0.017%
Inbound NAT bucket not found	173	0.007482%
Inbound gateway not connected	78	0.003373%
Inbound bad protocol	57	0.002465%
Inbound source IP incorrect	30	0.001297%
Inbound destination IP incorrect	6	0.0002595%
Inbound ping accepted	3	0.0001297%
Lost on wire	2	0.0000865%
Outbound rewrite	1	0.00004325%

Other entries in this table deserve some discussion. The next most common reason, incorrect sequence numbers, indicates that ARG's replay protection mechanism blocked packets. This commonly occurs at initialization, resulting in an average of 4 sequence number issues per test over the course of 143 tests, one for each gateway. The third entry on the list, "Inbound Unwrapped," occurs at the end of a test, where the receiver of a packet

is shutdown before the sender. The results processor traces packets through the network using a combination of packet captures and text log files of actions on each host. When a test run ends, these utilities shutdown sequentially. If a host sends a packet to a gateway that is still recording packet captures but has already shutdown its application-level receiver script (which creates the text log files), the test processor returns the last message it saw regarding the packet, which is typically whatever action the gateway took. Appendix F covers the test processor in more detail. The end-of-test issues are also the case for the "outbound rewrite," "ping accepted," and "unknown" entries. The remainder of the reasons are self explanatory.

A resampling of the data between the two hop intervals used in the basic tests gives a p-value of 0.639, indicating little significant difference between the two hop intervals. This confirms that hop interval in and of itself does not affect losses on the network, an important fact when considering the minimum hop interval in Section 5.2.

5.1.2 Invalid Packet Loss.

Traffic Flows 5 through 8 include invalid traffic that ARG should reject and hence should be "lost." Table 5.2 displays the percentage of invalid traffic rejected (with 95% CIs), clearly revealing ARG has an extremely low false negative rate. As shown, over the course of 28 repetitions only one test appears to have allowed a single packet through.

Table 5.2: Basic tests, packet loss of invalid traffic

Flow	**Mean**	**CI**	**Replications**
Flow 5	100%	(100%, 100%)	28
Flow 6	100%	(100%, 100%)	28
Flow 7	100%	(100%, 100%)	28
Flow 8	100%	(99.99%, 99.99%)	28

Flow 8 shows a small deviation from the expected 100% packet rejection, as 1 of 746 packets made it through, but further examination of this number shows a rare post-processing problem. If two identical packets are sent at the same time, the test run processor leaves the second one unmarked, which is interpreted later as successfully received. The problem lies with the log analyzer, not ARG itself, which did in fact reject every packet. This result offers convincing evidence that ARG effectively blocks unexpected inbound traffic, but it gives no indication of its suitability against more focused attacks.

Table 5.3: Basic tests 5-8 packet rejection reasons. 112 tests with 1,268,746 total packets represented.

Reason	**Count**	**% of Total Packets**
Inbound NAT bucket not found	60,884	4.799%
Inbound unwrapped	687	0.05415%
Outbound message too long	486	0.03831%
Unknown	340	0.0268%
Outbound sequence number incorrect	242	0.01907%
Inbound source IP incorrect	14	0.001103%
Inbound ping accepted	5	0.0003941%

Additionally, Table 5.3 illustrates that packets are rejected via the NAT table quite frequently over the 112 test runs with invalid traffic. This operation costs the gateway minimal processing time, as it can reject after a single hash table lookup. Fast decisions on packets lower the possibility of an effective Denial of Service (DOS) attack, as the Central Processing Unit (CPU) can continue to transform valid traffic. Section 5.1.1 discusses the meaning of the remainder of this table.

5.2 Minimum Hop Interval

The minimum hop interval sequence of tests measures the change in packet loss at specific latencies as the time between hops decreases. For these tests, a fixed packet rate and Flow 4 are always used. Section 4.8 covers the specifics of these tests. The figures below document the results of the tests, broken up by traffic type and direction.

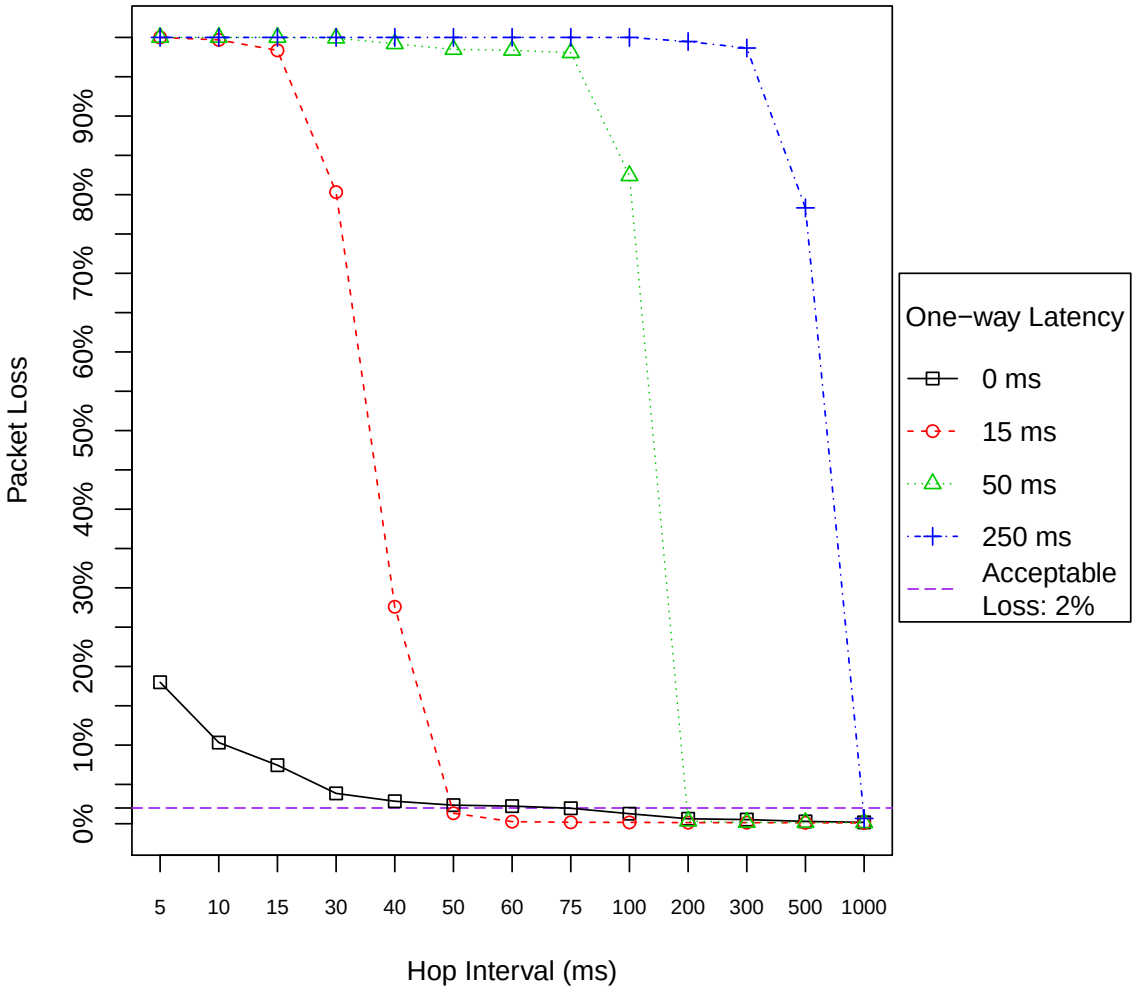


Figure 5.3: Hop interval tests, packet loss of UDP and TCP traffic between ARG networks. Note: X axis is not linear.

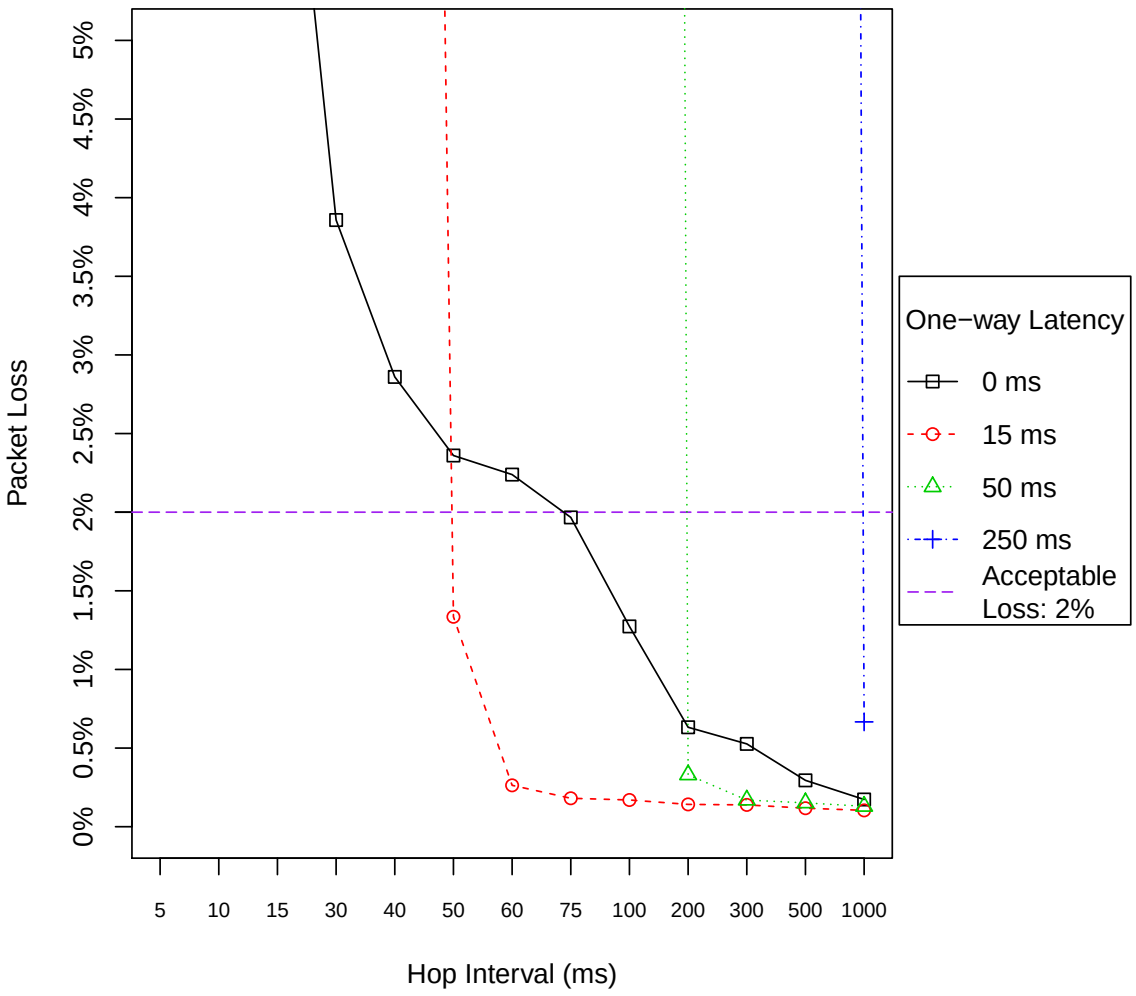


Figure 5.4: Hop interval tests, scaled view of packet loss between ARG networks

Figure 5.3 displays the results of the hop interval tests, separated by network latency. Figure 5.4 displays the same numbers, but with a much tighter Y scale to reveal differences at large hop intervals. The numbers shown here only include packets sent between ARG-protected clients; traffic to the external host and administrative packets between gateways are not included (packets to external hosts are considered separately later). The highest

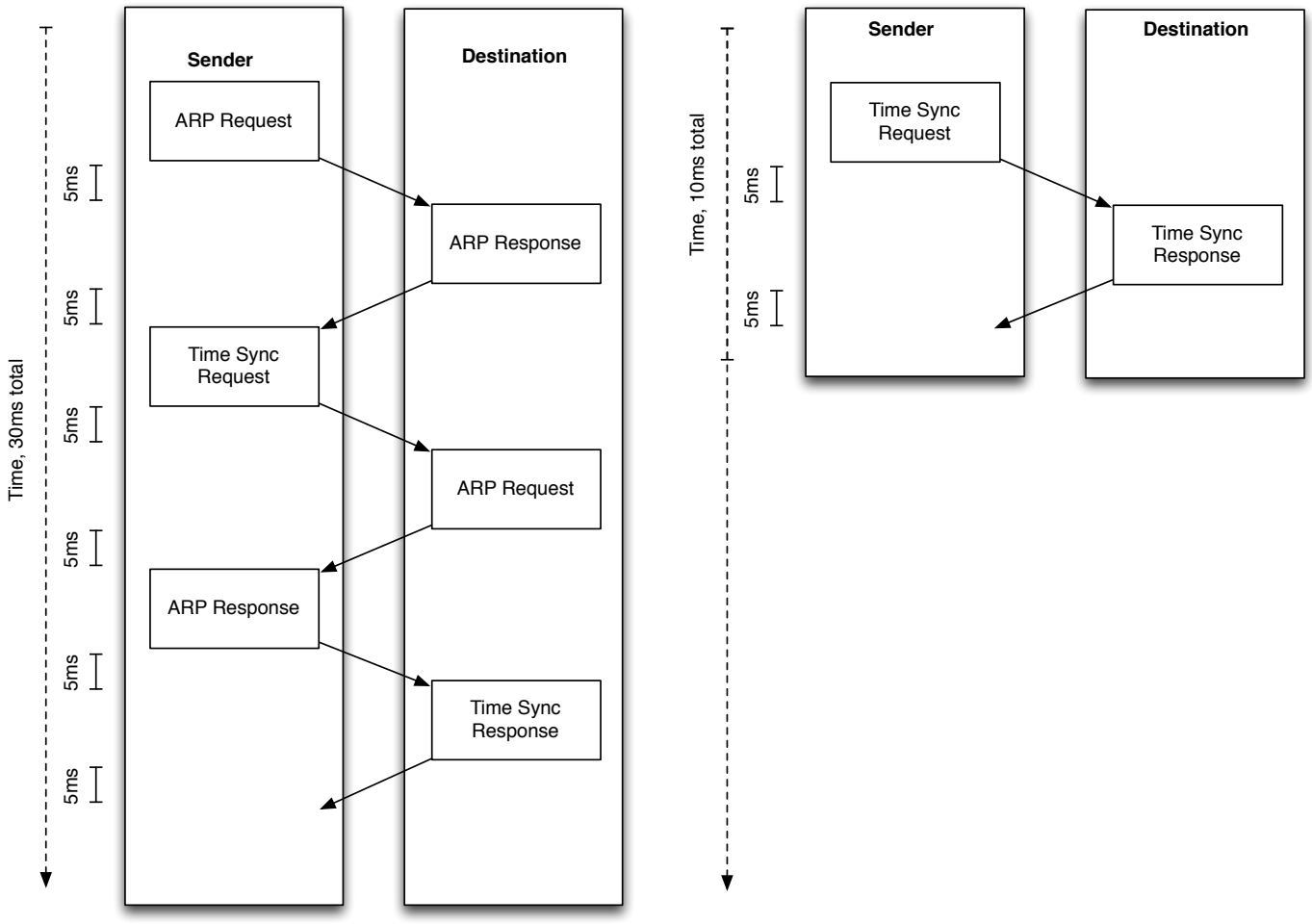
standard deviation in any of the hop interval-latency groups is 2.2238, implying the potential for significant (but still potentially acceptable) variation in network performance.

Interestingly, the zero millisecond latency trials exhibit a noticeably different decline than the others, with zero milliseconds giving a relatively steady decline in packet loss and the others remaining relatively flat before a rapid drop. The 15 millisecond latency curve gives a hint the shapes may not be as different as they appear, as it shows two or three mid-way data points on the way down to acceptable packet loss. With tests against more hop intervals, these curves would likely show a more gradual (although still rapid) decline.

Chapter 4 hypothesizes that packet loss reaches acceptable levels when the time between hops is equal to (or exceeds) the one-way network latency. Theoretically, this time frame should allow a packet to cross the network from one gateway to another before the receiver changes IP addresses twice, even if the packet is sent just before the receiver hops the first time (as mentioned in Chapter 3, gateways accept packets at the current and previous IP). Any shorter hop interval results in a period of time before each hop that guarantees sent packets arrive too late. Figure 5.4, however, shows that loss drops below 2% when the hop interval is around four times the latency. This difference from the anticipated behavior is likely the result of the test environment.

As Section 2.1.3 discusses, Ethernet uses ARP requests and responses to determine the source and destination for packets on the local network. In the test environment, the gateways and the external host are all on the same local network. This results in extra Ethernet ARP requests and responses, when compared to operation over the Internet. On a normal Ethernet segment with minimal latency (the zero millisecond test), the ARP process takes minimal time and has little impact on operation. To speed the process even further, ARP data is cached, so when a system wants to send to the same IP again, it references the ARP cache table and does not have to go through the request-response process again. With short hop intervals, however, gateways frequently need to send to different IP addresses and

therefore must send and receive the requisite ARPs very frequently (generally, once every hop). When the network latency is not zero, these additional frames can cause significant delays. Figure 5.5 illustrates what happens when a time synchronization between gateways occurs with a five millisecond one-way latency.



(a) With 5 millisecond one-way universal latency, including ARPs, as occurs on the test network.

(b) With 5 millisecond latency across the network and insignificant ARPs (not shown), as might occur in real operation across the Internet.

Figure 5.5: Time synchronization process, including ARPs

If there is no need for these ARP requests (or they were negligible, time-wise), the entire time synchronization process completes in 10 milliseconds. Because ARPs on the test network experience the same delay as UDP and TCP packets, however, it takes 30 milliseconds to synchronize times. The time base for the other gateway is calculated correctly despite the extra delay because the latency of that particular packet is taken into account. Later packets may take a wide variety of latencies, however, because they may not require any ARPs, only one direction might require it, or both directions may require it. For a 5 millisecond one-way latency, this leads to variations of 5, 10, or 15 milliseconds, an acceptable and tolerable range. By the time tests get to set latencies of 250 milliseconds, one-way latencies range from 250 to 750 milliseconds, with RTTs of up to 1500 milliseconds. This degree of variation is difficult to compensate for, leading to significant packet loss until the hop interval exceeds three times the set latency. Additionally, at shorter hop intervals the triple-latency problem is encountered more frequently, as the ARP cache must be refreshed more often.

On the Internet, the triple-latency problem does not occur. An ARG gateway sending a packet to another gateway uses ARP requests to get the hardware address of the local router, a process which takes under a millisecond on most Ethernet networks. Once the ARG gateway has the hardware address of the router, it sends the data packet out to the other gateway. After the packet leaves the local network, IP address hopping does not affect network operation at all, resulting in the packets between gateways experiencing same latency as any other data on the network. A 30 millisecond one-way latency across the Internet (or other Wide Area Network (WAN)) means packets genuinely take (on average) 30 milliseconds to reach their destination, whereas a 30 millisecond latency on the test network often means packets take 90 milliseconds.

Despite having no latency, the zero millisecond test exhibits high loss until around 75 milliseconds hops, similar to when the 15 millisecond latency test indicates a usable

connection. This may indicate that hopping more frequently than every 50-75 milliseconds is impractical regardless of network conditions with the current implementation. It may be possible for the sender of a packet to calculate "future" IP addresses to use for each packet sent, so that the addresses are current when they arrive at the destination. This is an area for future research, as Section 6.3 discusses. Previous research into address space randomization and IP hopping limit themselves to hopping on the order of minutes or hours and never explore the maximum rate possible.

ARG appears to have little additional impact on TCP, when compared to UDP. Figure 5.6 displays losses associated with only TCP packets travelling between gateways. Compared to Figure 5.3 (which includes UDP and TCP packets), trends appear similar.

Table 5.4 shows the packet rejection reasons for all hop interval tests. As expected, far and away the most common rejection reason for this set of tests is incorrect source and/or destination IPs. Other entries in this table are discussed in Section 5.1.1.

Finally, Figure 5.7 demonstrates there is little correlation between the hop interval and the flow of traffic to external hosts. Over all 612 hop interval test runs, packet loss to and from the external host never exceeds 0.059%. This makes sense, as the changing IP is only referenced when the NAT module first creates a table entry. Later packets on the same connection consult the table information and are unaware of any IP address changes.

5.3 Maximum Packet Rate

The final numerical test performed against ARG measures the maximum packet rate it is capable of supporting. For these tests, latency is set to 20 milliseconds and Flow 4 is used. Hop intervals vary between 50 ms and 500 ms to check if this has an effect on the supported packet rate. Most importantly, traffic generators are run with steadily increasing send rates. Section 4.8 contains more details. Because throughput varies somewhat on each run, the data is clustered into the groupings listed in Table 5.5 and colored in Figure 5.8.

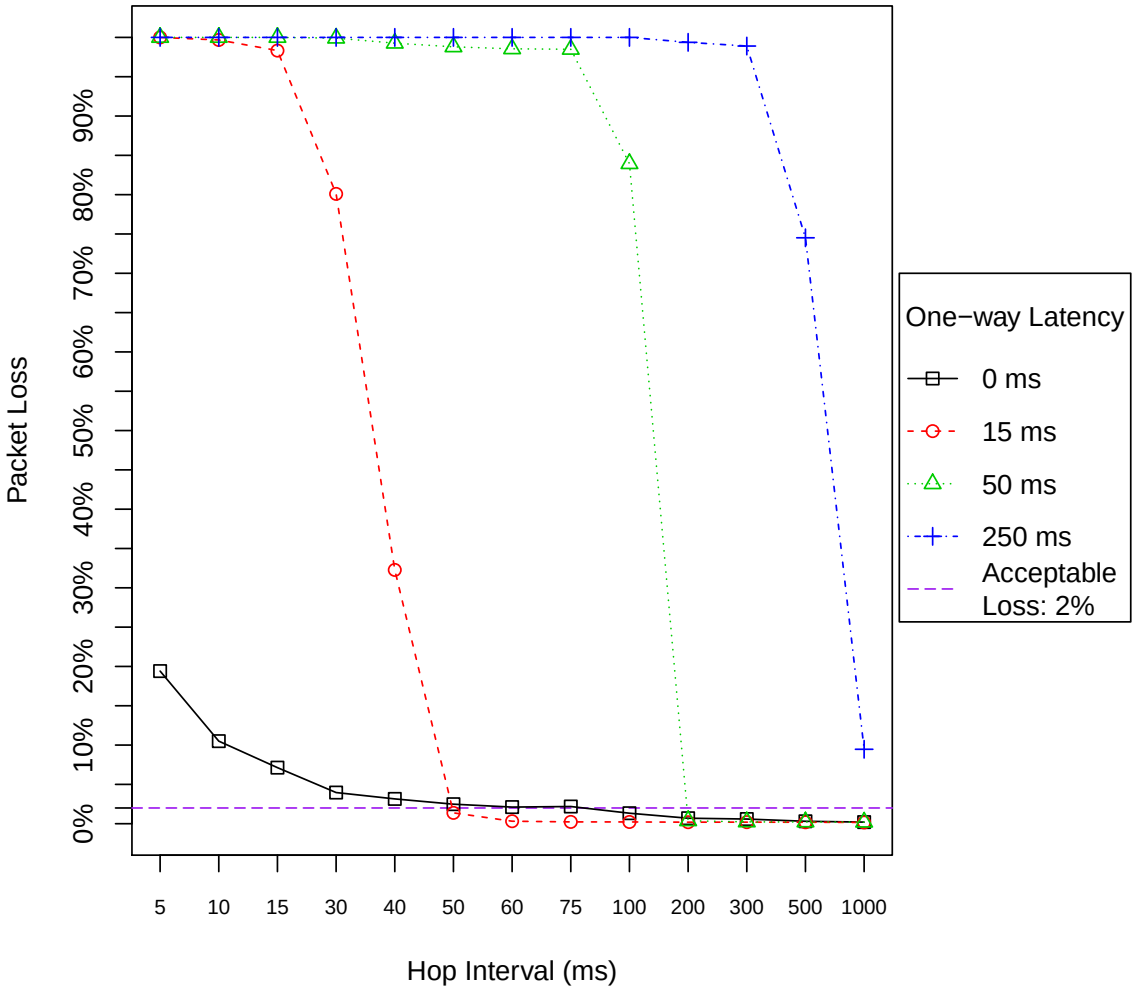


Figure 5.6: Hop interval tests, packet loss of TCP traffic between ARG networks

Figure 5.8 shows the raw data from these tests, with colors indicating the 7 K-means clusters, grouped by bit-wise throughput. Clusters are chosen via the Hartigan and Wong algorithm, with 500 random starts. This graph includes all packets on the network. Visually, the plot reveals fairly similar loss in each cluster, although some clusters exhibit

Table 5.4: Hop interval test loss reasons. 612 tests with 15,387,651 total packets represented.

Reason	**Count**	**% of Total Packets**
Inbound destination IP incorrect	506,075	3.289%
Inbound source IP incorrect	165,656	1.077%
Inbound unwrapped	4,935	0.03207%
Unknown	4,235	0.02752%
Outbound message too long	4,106	0.02668%
Outbound sequence number incorrect	2,180	0.01417%
Outbound gateway not connected	1,000	0.006499%
Inbound NAT bucket not found	110	0.0007149%
Inbound bad protocol	94	0.0006109%
Inbound gateway not connected	14	0.00009098%
Outbound wrapped	12	0.00007798%
Inbound ping accepted	10	0.00006499%
Inbound connection data received	1	0.000006499%

a fair amount of variation. A Tukey test with each cluster confirms this result, as shown in Figure 5.9.

On Figure 5.9, numbers along the side correspond to the cluster numbers shown in Table 5.5 and Figure 5.8. Six means stand out as being distinctly different: 1-4, 2-4, 3-4, 4-5, 4-6, and 4-7. Cluster 4 is statistically different from all other clusters, with 95% confidence. However, the other data gives reason to believe that throughput was not the cause of loss increase and variation. No changing trend is seen in any of the surrounding throughputs' losses (either up or down) and faster throughputs (Clusters 5 through 7) show

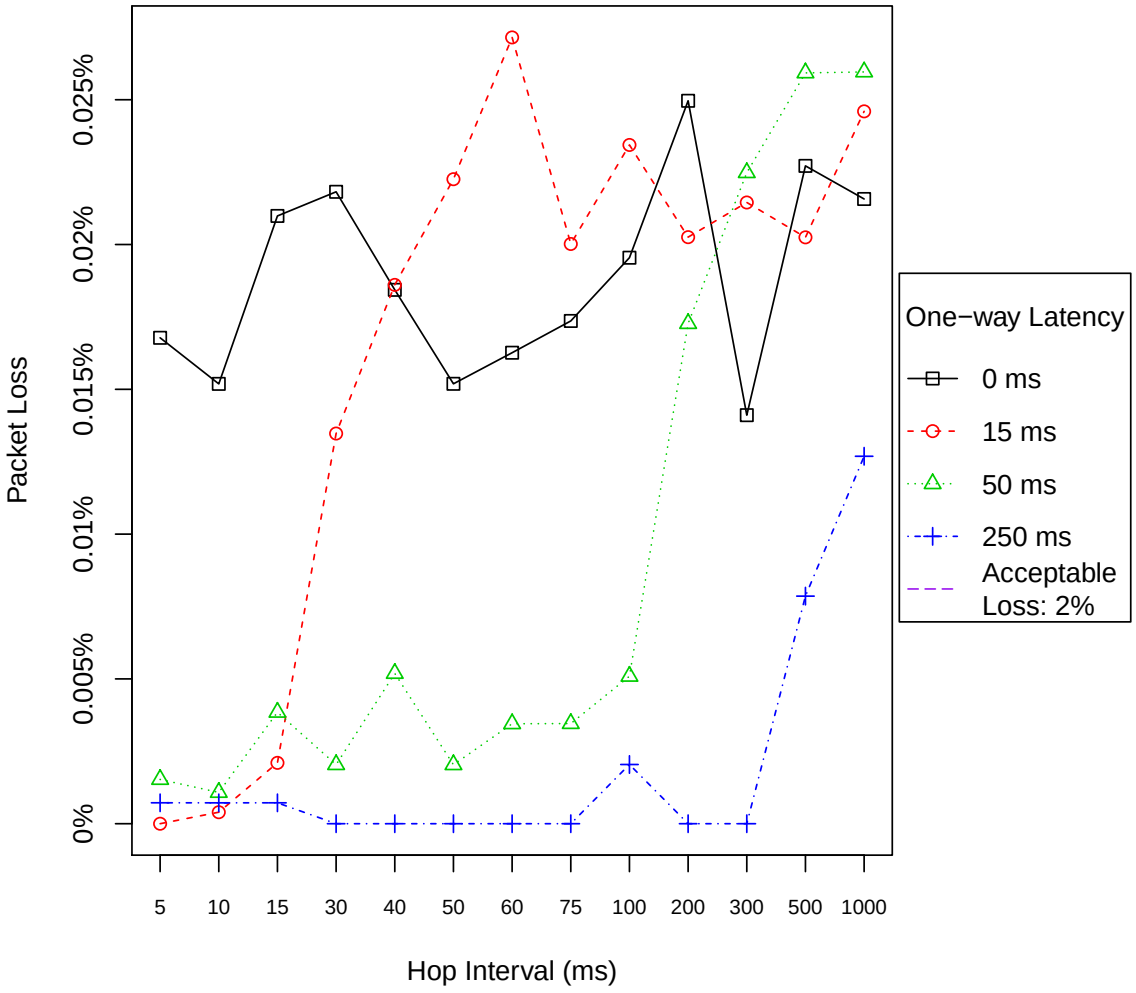


Figure 5.7: Hop interval tests, packet loss of externally-bound traffic

loss comparable to the slower throughputs (Clusters 1 through 3). While the precise cause of the increased loss in some of these tests is unknown, the amount of data the gateways handle is not believed to be the cause.

The results of these tests are limited to just the implementation of ARG and imply only a minimally possible throughput. Despite these limitations, ARG performs well with

Table 5.5: Packet rate clusters

Clust	Mean (Kbps)	Min (Kbps)	Max (Kbps)
1	485.1	300.3	573.7
2	854.5	782	980.3
3	1401	1190	1581
4	2223	1969	2421
5	3250	3243	3264
6	3796	3778	3814
7	4374	4351	4389

reasonably rapid traffic of over four Mbps, giving no reason to believe it is not capable of higher rates. Time constraints prohibit more rapid throughput tests.

5.4 Fuzzing Test

The final series of tests run on ARG consist of sending invalid and replayed traffic to the gateways. The primary objective is to verify that ARG remains stable and able to pass valid traffic while under attack.

The custom run-processing tool this thesis uses to generate packet loss statistics is unable to process the logs from the malicious traffic generators, making it difficult to determine precisely what occurs in each run. This calls into question the accuracy of the results from the run processor. Nonetheless, the tool reports a mean packet loss between 21.8% and 23.5% (with 95% confidence), which indicates ARG experiences significant difficulty when faced with malicious traffic. A manual examination of log files does seem to indicate this is the case, with some traffic successfully traversing the network but a huge amount being rejected.

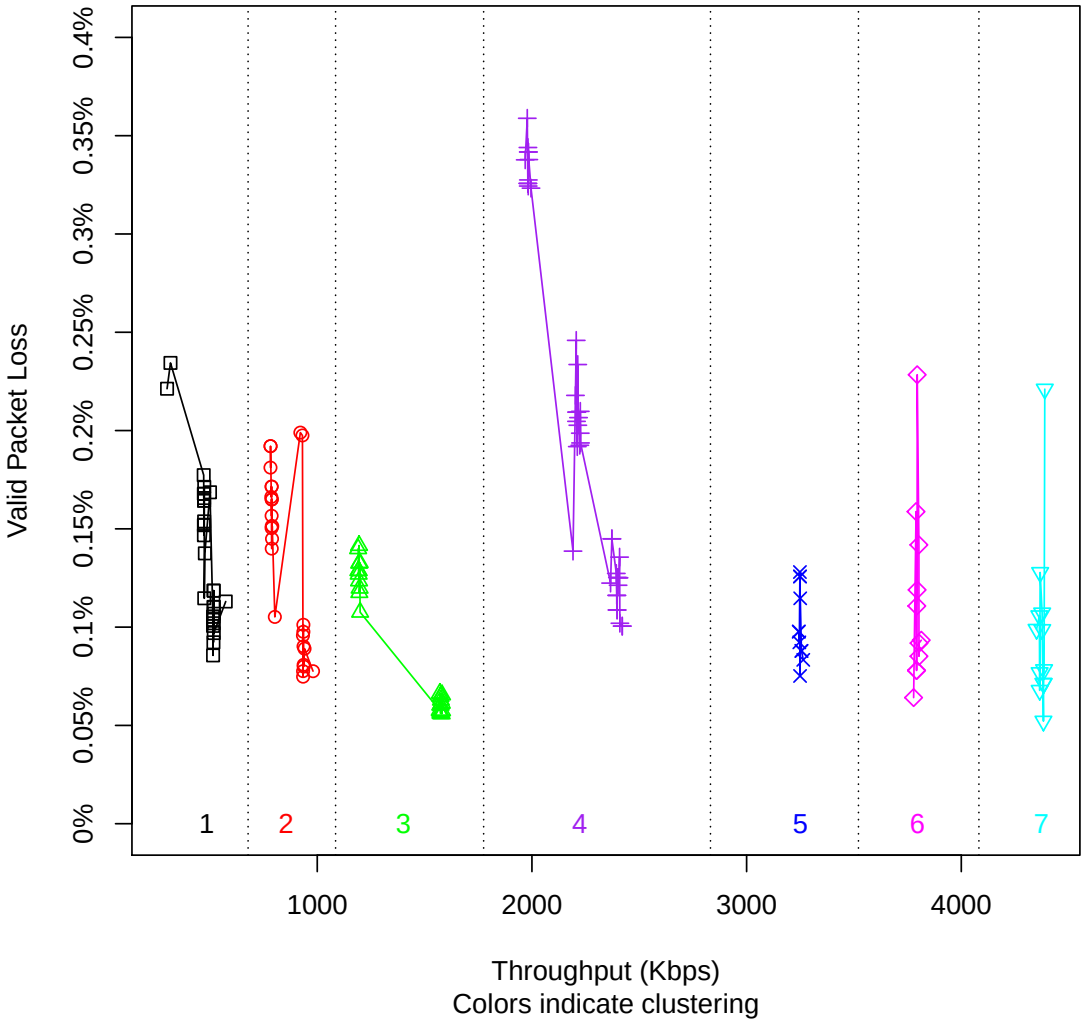


Figure 5.8: Packet rate tests, clustered loss verses throughput

Despite the high losses, ARG does appear to remain stable throughout the barrage. In all cases, ARG never crashes and maintains around the same ability to handle valid traffic (that is, although the losses appear high, they remain consistent throughout). Further research is needed to identify the causes behind the high losses, but the problem likely lies

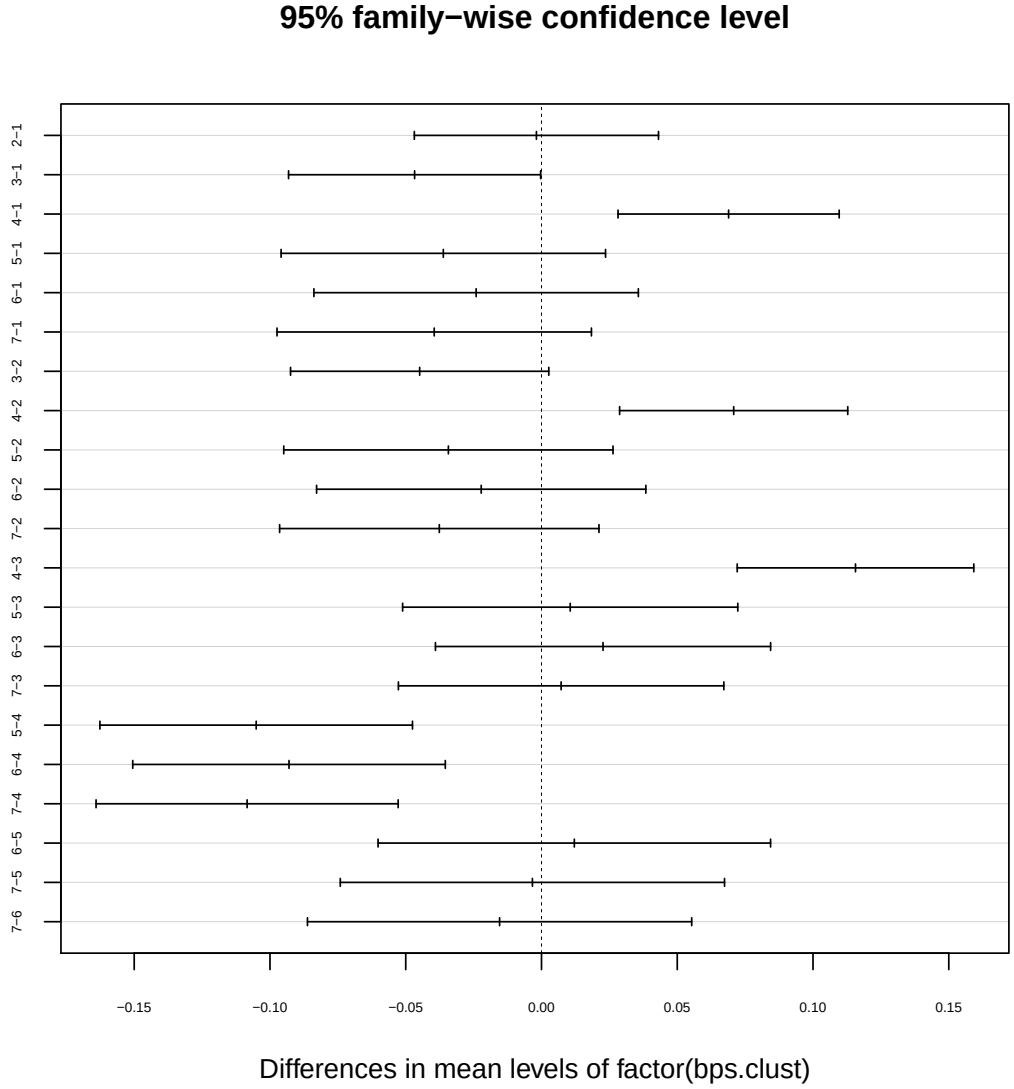


Figure 5.9: Packet rate tests, Tukey test against clustered data

in implementation and not architecture. Even if ARG crashes, traffic flow entirely stops, preventing the attacker from penetrating the network.

5.5 Overall Analysis

Chapter 4 presents several questions that this research attempts to answer. Each is revisited below along with an accompanying conclusion. More details on each answer may be found in earlier sections of this chapter.

1. Does ARG classify traffic correctly?

   Overall, ARG does classify traffic correctly. Throughout the testing, no actual false negatives occurred, indicating ARG's NAT system is robust and provides a solid way to classify invalid inbound traffic. However, the traffic used to test this system is fairly benign and does not intentionally attempt to circumvent ARG. Testing needs to be done with a more intelligent threat.

   For expected traffic on the network, whether it is between ARG-protected networks or to external hosts, ARG is reasonably accurate. In basic tests involving a range of traffic types, ARG averaged less than 0.1123% loss of valid, expected traffic. This falls well below Chapter 4's definition of "acceptable loss" of 2%. Based on this, ARG accurately classifies traffic.

2. What is the maximum packet rate ARG can handle?

   The testing done for this research did not find an upper limit on the amount of traffic ARG could handle, with the gateways showing the capability of handling over 4.3 Mbps with no statistically significant change in packet loss.

3. What is the minimum supportable time between hops? How does latency affect this?

   When ARG operates on a network with a one-way latency less than 15 milliseconds, hops may occur every 50 to 75 milliseconds and still maintain a viable communication channel (losses are less than 2%). Beyond this point, packets begin flowing smoothly when the time between hops exceeds four times the latency. However,

there is reason to believe hops could be faster in a real-world deployment, due to the test network factors Section 5.2 discusses. This is a significant contribution because previous research focuses on address changes on the order of minutes or hours, rather than multiple times a second.

Realistically, network latencies vary wildly. However, ARG allows flexibility in setting the hop interval for every gateway separately and could be easily expanded to allow gateways to change the hop intervals they use between each other while leaving the rate used for others alone. Section 6.3 discusses this in more depth. This flexibility allows networks with low-latency networks to hop frequently and high-latency networks to hop more slowly. The results here provide guidance on the maximum hop interval to chose based on network conditions.

4. Is ARG stable when presented with corrupt, malformed, or replayed packets?

   ARG is stable in the face of bad traffic. Long-running fuzz tests do not cause ARG to crash, nor do they permanently break traffic flow. However, malformed traffic does appear to have an impact on valid packet loss, based on both automated analysis and a manual inspection. The cause of this needs exploration to determine the correct fix.

5.6 Summary

This chapter analyzes the results for each test (Figure 4.3) sequence, each broken out into its own section. An overall analysis in context of the original research questions is then presented, summarizing the findings of this thesis.

VI. Conclusions and Recommendations

This chapter summarizes the work and findings of this research. Section 6.1 summarizes the conclusions reached. Section 6.2 discusses the impact of this research. Section 6.3 provides recommendations for future work in the network address space randomization area in general and on ARG itself.

6.1 Research Conclusions

This research demonstrates that IP hopping is a suitable method of blocking unexpected external traffic while maintaining a minimal false-positive rate. This can be done completely transparently from the perspective of the internal and external hosts; the tool this thesis presents works with no configuration changes to any host other than the gateway.

In addition, ARG proves rapid IP address changes are possible, with network latency as the primary limiter. Tests demonstrate that—under this implementation—IP addresses may change around 15 times per second (changes every 50 to 75 milliseconds) and still allow for reliable communication.

ARG also demonstrates good throughput, a critical aspect of usability to a real network. Test rates reach four Mbps with no indication that ARG is unable to handle greater rates. Running a fuzzer against ARG found that, while gateways themselves remain stable in the face of malformed traffic, the attack may have an impact on connectivity and valid packet loss.

6.2 Research Impact

This thesis presents a new IP address hopping tool that combines features of previous efforts in this area. Through a gateway-based solution, ARG avoids requiring changes to existing network architecture or any clients inside. ARG applies IP address changes to all

packets entering and leaving the network and packets between ARG-protected networks include full encryption and authentication.

Of primary importance to this field of research is the demonstration that IP address changes may occur multiple times per second. Previous research focuses on changes on the order of minutes or hours and may kill on-going connections when address changes occur. ARG's design allows for connections to persist across hops without participation of either end of the stream, ultimately allowing for much more frequent address changes and a potential amplification of the benefits of address space randomization.

6.3 Future Work

6.3.1 IPv6 Support.

IPv6 support is slowly becoming a requirement for any network system. For an IP hopping system, IPv6 offers the benefit of a greatly increased address space, allowing systems to hop in a much broader range of addresses. ARG is entirely IPv4 in its current implementation and cannot transport IPv6 packets to external hosts or to other gateways.

6.3.2 Fragmentation Support.

ARG currently has neither support for fragmenting packets as they pass through the gateway nor for notifying the sender that fragmentation is needed. Packets to and from external hosts pose no problem, as the original sender will handle this themselves. However, packets between gateways/ARG-protected networks have additional data added, potentially exceeding the maximum transmission unit of the network. In this case, ARG has no way to recover and the packet is permanently dropped without notice. A more complete implementation should notify the sender that fragmentation is needed.

6.3.3 More Extensive Malicious Testing.

Due to time constraints, a full battery of robust malicious tests could not be performed against ARG. As demonstrated by the basic fuzz testing, ARG handles errors without becoming unstable, but may lose additional packets. The reasons behind this potential

issue needs more exploration to determine the root cause and what should be done to fix it. More extensive work in both undirected (i.e., fuzz testing) and directed attacks is needed. For example, malicious hosts might attempt to falsely connect to a gateway or perform replay attacks in a more intelligent manner.

6.3.4 More Intelligent NAT.

ARG currently blindly opens holes in the NAT when it sees outbound packets and closes them after seeing no activity for a fixed amount of time. A transport layer examination would allow more fine-grained NAT work, by watching for actual connection establishment and teardown packets.

6.3.5 Integration with Other Defenses.

Network defenses often perform better when working in tandem. ARG has the potential to detect certain types of probes into the network. If this information can be passed off to an IDS, it might alert an operator or take other defense actions on the network. In an even more active approach, ARG might work with a honeypot to present a fake view of the network to an attacker. By examining what systems an attacker probes, it might be possible to determine the identity of the adversary, their goals, and their intended target in the network, all valuable information to those defending the network.

6.3.6 Latency Compensation.

The current design of ARG exhibits problems transferring packets when the hop interval is less than four times the latency. This situation is easy to detect, as latency between the gateways is already being calculated. To enable more flexibility with varying network conditions, gateways could change their hop interval on an individual basis to match the network conditions to every other gateway with which they connect. For example, say there are gateways `GateA`, `GateB`, and `GateC`. The latency between `GateA` and `GateB` is 50 ms, the latency between `GateB` and `GateC` is 40 ms, and the latency between `GateA` and `GateC` is 200 ms. By default, they each hop every 200 ms. When they

perform time synchronization, `GateA` and `GateC` would detect the high latency and change the hop interval for *just* each other to 600 ms, but leave their hop interval at 200 ms for `GateB`.

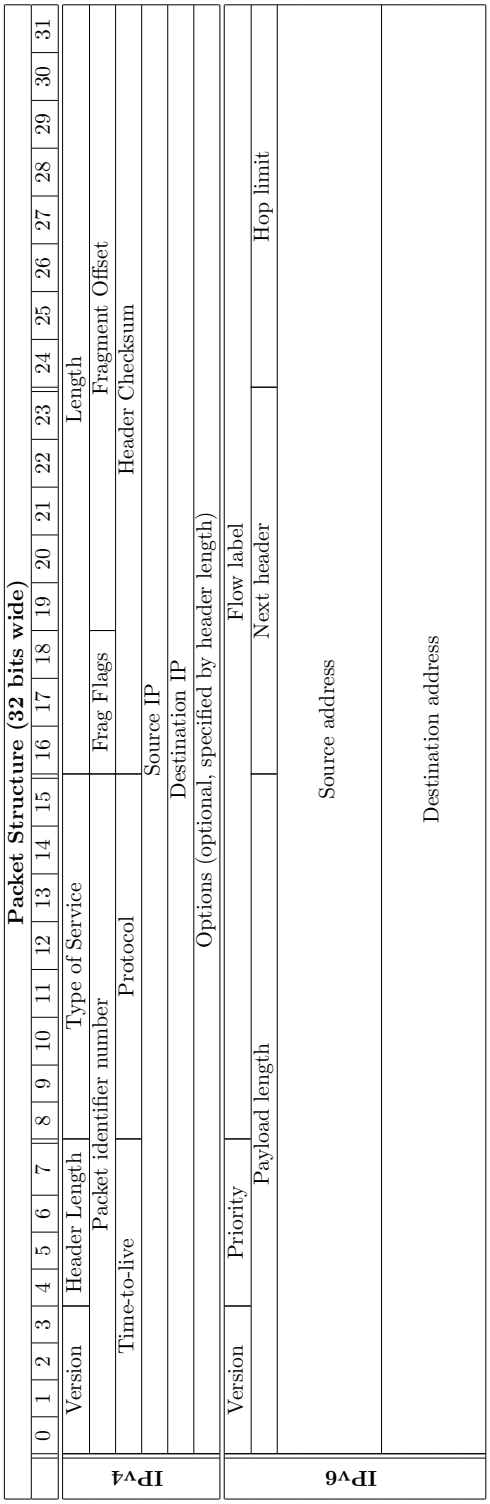
Alternatively (or perhaps in addition), it would be possible to send packets with IP addresses "in the future," so that when they arrive at their destination the addresses would be current. That is, when a gateway is about to send a packet to another gateway, it calculates the addresses based on the current time + latency, rather than just the current time. This relies on network latency being relatively stable, as sudden drops in the latency would cause packets to contain future addresses when they arrive at the receiver.

6.4 Summary

This chapter reviews the work and findings of this thesis. The impact of the research is discussed and recommendations for future work are given.

Appendix A: IP Packet Structure

Table A.1: IP packet structure



Appendix B: ARG Protocol

This appendix details the format of each ARG message type. Section B.1 presents the structure of each message type. Section B.2 gives the steps of each exchange type and their effect on the gateway's state.

B.1 Message Formats

The base header for the ARG protocol is given in Table 3.4. Four possible payloads are delivered under this header: `WRAPPED`, `PING`, `CONN_REQ`/`CONN_RESP`, and `TRUST_DATA`. The format for each is shown in individual sections below.

B.1.1 WRAPPED.

Use: Transfer packets between ARG-protected networks.

This message type contains no additional information. After the original packet is encrypted, it is used as the payload to the ARG header as-is.

Table B.1: Data in ARG `WRAPPED` message

Data	**Size**	**Data Type**
Packet Data	-	Raw, encrypted packet

B.1.2 PING.

Use: Time synchronization between two gateways. See Section B.2.2.

Table B.2: Data in ARG PING message

Data	Size	Data Type
Request ID	4 bytes	Unsigned Integer
Response ID	4 bytes	Unsigned Integer
Time Offset	4 bytes	Unsigned Integer

B.1.3 CONN_REQ/CONN_RESP.

Use: Connect to other gateways.

Note that these message types are formatted identically. The only difference is the message type number used, as this allows gateways to determine when it is a request for new data or just a response to a previous request.

Table B.3: Data in ARG CONN_REQ and CONN_RESP messages

Data	Size	Data Type
Symmetric Key	32 bytes	Raw
Initialization Vector (IV)	32 bytes	Raw
Hop Key	16 bytes	Raw
Hop Interval	4 bytes	Unsigned Integer

B.1.4 TRUST_DATA.

Use: Allow on-the-fly addition of new gateways.

Table B.4: Data in ARG TRUST_DATA message

Data	Size	Data Type
Gate Name	10 bytes	Null-padded String
Base IP	4 bytes	Unsigned Integer
Mask	4 bytes	Unsigned Integer
N	130 bytes	Raw
E	10 bytes	Raw

B.2 Protocol Exchanges

The steps for each exchange are given in the sections below. In the following descriptions, local is the initiating gateway in a given exchange and remote is another gateway with which it is communicating.

B.2.1 Connect process.

1. Local sends CONN_REQ containing its hop key, hop interval, and symmetric key.

2. After validating the packet, remote saves the connection data. If time synchronization data is available for local, then remote marks it as connected. If it is not available, remote schedules a time synchronization request.

3. Remote sends CONN_RESP acknowledgment back, containing its own hop key, hop interval, and symmetric key.

4. Local receives, validates, and saves the data from remote.

5. Local marks the remote gateway as having connection data and marks remote as connected or schedules a time sync, as appropriate.

B.2.2 Time synchronization.

1. Local sends `PING` containing random 4-byte unsigned integer in the `Request ID` field (see Table B.2), `null` (0) in `Response ID`, and its time offset in `Time Offset`, which is the difference between the current time and its base time.

2. Local notes the time it sent the request.

3. Remote validates the message and responds with a new `PING`, giving a new random request integer, the received response int (from local) set to the request int, and its own time offset.

4. Local ensures received response integer matches the request integer it sent.

5. Local determines the connection's round-trip latency from the send time, then remote's time base is calculated based on half of this. That is,

$$\text{remote time base} = \text{received time offset} - \frac{\text{latency}}{2}$$

   This value is saved and used in IP calculations for the remote gateway in the future.

6. Local marks the remote gateway as having time sync data available and, if connection data is available, remote is marked as connected. If connection data is not available, local schedules a connect process.

7. Local sends a response to remote, with `Request ID` set to 0, `Response ID` set to the value remote sent in its request, and `Time Offset` set as in step 1. This second `PING` send from local is necessary because remote does not know the latency of the initial request.

8. Remote receives and validates the final response, saves the data, and marks local as connected if connection data is already available (or schedules a connection data request if needed).

B.2.3 Trust Data Exchange.

1. For each gateway it knows about, local sends a `TRUST_DATA` packet to remote, containing the gateway name, base IP, IP mask, and public key to the remote gateway. Each packet covers one gateway, with no data about the local or remote gateways included.

2. Remote validates the message, then adds the data in each `TRUST_DATA` packet to their list of gateways (if they do not already have it). At this point the new gateway appears just like one read in from a configuration file.

3. Within two seconds, remote attempts to connect to the new gateway, just as they would any other gateway they had not yet successfully contacted.

B.2.4 Route packet.

1. Local receives a packet on its internal interface.

2. Local takes the outbound packet and encrypts with with remote symmetric key. Local determines the destination gateway based on IP range.

3. Local sends WRAPPED message to remote current IP with the encrypted packet included. An HMAC of the packet (encrypted data and headers) is included in the header.

4. Remote receives and validates the message.

5. Remote sends the original, decrypted packet into the internal network.

Appendix C: ARG Testing

This appendix covers the steps used to run a test. The events shown below occur in order for each test.

1. Set time on all hosts to be the as similar as possible (used for post-processing only).
2. Set artificial network latency on external interfaces of gates.
3. Start `tcpdump` on each host. Gateways have two instances started, one for each interface. The exact commands with the appropriate traffic filters appear in Table C.1.

Table C.1: Test run `tcpdump` calls

Host	**Command**
Gateway	`sudo tcpdump -i eth2 -w gateX-inner.pcap -n ip and not arp`
	`sudo tcpdump -i eth1 -w gateX-outer.pcap -n ip and not arp`
Prot/Ext	`sudo tcpdump -i eth1 -w clientX.pcap -n ip and not arp`

4. Set ARP cache size on gateways and external host to allow for 65,536 entries. This is needed at short hop intervals only because all systems are on the same network segment.
5. Push configuration files for ARG. `ProtA1` and `ProtC1` each know about `ProtB1`, but not about each other. `ProtB1` is given configuration files for both `ProtA1` and `ProtC1`.
6. Start ARG on the gateways.

7. Start traffic generators on hosts, as appropriate for the test being run. See Section 4.8 for general flow of traffic for each test type.

8. Wait for the test to finish. For all tests discussed in this thesis, tests run for five minutes.

9. Stop traffic generators.

10. Stop ARG.

11. Stop traffic collectors (`tcpdump`).

12. Retrieve log and pcap files from every host into a directory.

After the logs are collected together, the run is processed by a separate script, `process_run.py`. See Appendix F for details on its use.

Appendix D: ARG Building and Configuration

This appendix documents everything needed to use ARG. Section D.1 covers the build environment needed to build ARG from source. Section D.2 documents calling ARG and creating the necessary configuration files.

D.1 Building

ARG runs on Ubuntu 12.04 and 12.10. Other versions and distributions are untested, although they may work if the below requirements are met.

D.1.1 Required Packages.

- gcc >=4
- linux-headers 3.5.0
- Autoconf >=2.69
- Automake >=1.11
- libtool >=2.4.2
- libpcap-dev >=1.3.0
- libpolarssl-dev >=1.1.4
- libpthread-dev >= 1.1.1

In Ubuntu:

```
$ sudo apt-get install automake autoconf build-essential \
   libtool libpcap-dev libpolarssl-dev
```

D.1.2 Compilation.

From the ARG source directory:

```
$ ./autogen.sh
$ make
```

This should produce two executables, `arg` and `gen_gate_config`.

D.2 Usage

D.2.1 Command Line.

ARG must be run as root. Usage is straightforward:

```
$ sudo ./arg <conf file>
```

A path to a configuration file is required. The path should point to the main configuration file, with supporting files in the same directory as the main one. See Section D.2.2 for more details.

ARG will start up and configure itself, then after a brief delay attempt to connect to other gateways for which it has configuration information. To end ARG's execution at any time, press `ctrl-c` to cleanly kill it or send it `SIGTERM` via `kill` (i.e. `sudo killall \ -SIGTERM arg`) to end it without cleaning up.

D.2.2 Configuration Files.

ARG requires at least three separate configuration files on start up, plus one for every gateway it should have initial knowledge about. The main configuration file may be called anything and contains four lines:

Listing D.1: main.conf

```
1 gateA
2 eth2
3 eth1
4 1000ms
```

In order, these are: gate name, internal network interface, external network interface, and hop interval. Hop interval *must* be given in milliseconds.

In the same directory as the main configuration file must be two files giving details on the local gateway called `<gatename>.pub` and `<gatename>.priv`. The private file (.priv) gives the full private key of a gateway, while the public file (.pub) contains the public key,

the base IP address of the gate, and the corresponding netmask. Examples of each are given in Listings D.2 and D.3.

Listing D.2: gate.pub

```
172.1.0.1
255.255.0.0
N = 9F6BA2C9BD3717D591B1E52256ACFB43DB2EDF010E0312611273\
    D4D327B215CFF87A1F04882559C49E61CA2D35E93B71D3950E4D\
    FA64A0F80D2B670D83C555D24E25EA38CD9BED14A8BD0FF05A36\
    649EDD64486E18521ABF695FA278A28303C50C89A91A4860A685\
    F961C45A4BE2CE3011F1B78C741FE9508595254DE336AF43
E = 010001
```

Listing D.3: gate.priv

```
N = 9F6BA2C9BD3717D591B1E52256ACFB43DB2EDF010E0312611273\
    D4D327B215CFF87A1F04882559C49E61CA2D35E93B71D3950E4D\
    FA64A0F80D2B670D83C555D24E25EA38CD9BED14A8BD0FF05A36\
    649EDD64486E18521ABF695FA278A28303C50C89A91A4860A685\
    F961C45A4BE2CE3011F1B78C741FE9508595254DE336AF43
E = 010001
D = 4C2A31A12ECC768FABC7115101962D89B2DB46E20B1EBE963029\
    B5019912854752508E272D20A32DC3F9B68D3917903606BF4D11\
    4652F370EF61D01F6DD846F2AC0137C22A0C33014860AC68F7DF\
    FC7F524B5D70FDA37530B06DFA518DCEDCF08537D4AB6DE5969C\
    13E143E6DF2C6DD3145B4E4FF9306BDCD209A3F66C5C3F81
P = E154E78FFFC51AAABA4BE0B5F9123366962612EBB8AB17008A0A\
    6367AB9C7A33ECEFCD807C8414CC56AC7678BE14604F13F9D66A\
    B754C26E91ED30B18EE76283
Q = B51E3A12CED999F234DEF316888AC624D65DFDCFA93DC40FABCF\
```

```
    70784057ADF27751A1B0103049AA0C4C89C92A081C774425F440\
    D5C4A4B18CE984E8B93EE441
DP = D159C9FECEDA78E93046F912F8C3014089B5FC1447B1A5A059A\
    04734F58B5F3A49238F6195CE3D68900A91D3A27E59F07E95BBB\
    1D07D0E5C1E7629AC7E21DA33
DQ = 9421B9BBA24464DDAD125FDD2125E7333FC4B60EFECB8EAC868\
    7EDE3DC341A07C24118ADE83FA63017490E34625529FAFDD8D0F\
    1AA24DFD27B7E8E7ECCEBBC41
QP = AA199F9F8C0EB3FEA2DB83EA22134576E9B9C8DD7C4B78D8CE4\
    2011E56EFAECF8A4C8AFE19F2F6E76B5B41C73D0B7F90B8FA1D0\
    04C2539060FAE762019D6F990
```

The first two lines of the public key file are, in order, the base IP address and mask for the gateway. The remainder of the parameters are hexadecimal integer values matching to parts of the RSA equations. The library used for encryption in ARG (PolarSSL) provides a straightforward way of reading these into the internal key structures. The exact details of each value are not overly important, as the public and private key files files can be produced quickly through the included `gen_gate_config` utility. This tool is built alongside `arg`, see Section D.1. Usage is:

```
$ ./gen_gate_config <name> <base ip> <mask>
```

This will produce a `<name>.pub` and `<name>.priv` file with the information given and a random public and private key.

For reference, `N` and `E` in both of configuration files come from the RSA encryption equation $c \equiv m^e \pmod{n}$. The values `D`, `P`, and `Q` in the private key file represent the original key generation values, as shown in the equations $n = pq$ and $d \equiv e^{-1} \pmod{\phi(n)}$. The remainder of the values are multiplied forms of the the previous values (e.g., DP is $D \times P$).

For every gateway that this gate should know about, another <othegatename>.pub should be placed in the directory. For instance, if gateA knows about gateB, then a gateB.pub file must exist. The information inside is the same format as its own .pub file. Look at Section D.3 for an example of the file structure and expected output.

D.3 ARG Execution Example

In this example ARG is being run on gateA and only knows about gateB.

Listing D.4: ARG Execution Example

```
1  $ ls
2  arg  conf
3  $ ls conf
4  gateA.pub  gateA.priv  gateB.pub  main-gateA.conf
5  $ sudo ./arg conf/main-gateA.conf
6  1355979615.731 LOG4 Starting at 20 Dec 2012 00:00:15
7  1355979615.731 LOG4 Reading from configuration file \
      conf/main-gateA.conf
8  1355979615.731 LOG4 Found public key for gate gateA
9  1355979615.731 LOG4 Found public key for gate gateB
10 1355979615.731 LOG4 Hopper init
11 1355979615.731 LOG4 Locating configuration for gateA
12 1355979615.731 LOG4 Configured as gateA
13 1355979615.731 LOG4 Generating hop and symmetric encryption \
      keys
14 1355979615.731 LOG4 Hop rate set to 1000ms
15 1355979615.733 LOG4 Hopper initialized
16 1355979615.733 LOG4 NAT init
17 1355979615.733 LOG4 NAT initialized
18 1355979615.733 LOG4 Director init
```

```
1355979615.733 LOG2 Internal device is eth2, external is eth1
1355979615.733 LOG4 NAT cleanup thread running
1355979615.733 LOG4 NAT Table empty
1355979615.735 LOG4 Using filter '(arp and not dst net \
   172.1.0.0 mask 255.255.0.0) or (not arp and src net \
   172.1.0.0 mask 255.255.0.0)' on eth2
1355979615.739 LOG4 Using filter '(arp and not src net \
   172.1.0.0 mask 255.255.0.0 and dst net 172.1.0.0 mask \
   255.255.0.0) or (not arp and dst net 172.1.0.0 mask \
   255.255.0.0)' on eth1
1355979615.739 LOG2 Internal IP: 172.1.0.0, external IP: \
   172.1.0.0, external mask: 255.255.0.0
1355979615.739 LOG4 Director initialized
1355979615.739 LOG4 Running
1355979615.739 LOG4 Starting connection/gateway auth thread
1355979615.739 LOG4 Connect thread running
1355979615.742 LOG4 Ready to receive packets on eth2
1355979615.765 LOG4 Ready to receive packets on eth1
1355979618.739 LOG4 Sending connect information to gateB
1355979618.740 LOG0 Outbound: Accept: Admin: connection data \
   sent: /p:253 s:172.1.194.123:0 d:172.2.151.79:0 \
   hash:8fb3b019948229847cd9e3adcd55fd90
...
^C
1355979940.504 LOG4 Director uninit
1355979940.541 LOG4 Director finished
1355979940.541 LOG4 Shutting down
1355979940.541 LOG4 NAT uninit
```

```
1355979940.542 LOG4 NAT finished
1355979940.542 LOG4 Hopper uninit
1355979940.542 LOG4 Removing all associated ARG networks
1355979940.542 LOG4 Hopper finished
1355979940.542 LOG4 Finished
```

Appendix E: Traffic Generators

This appendix covers the required environment for the two traffic generators and their usage. Section E.1 lists the required utilities to run the generators. Section E.2 discusses usage of each generator and the available command line parameters.

E.1 Environment

The traffic generators run on Ubuntu 12.04, Ubuntu 12.10, and Mac OSX 10.8. Other versions and distributions are untested, although they may work if the below requirements are met.

The following packages must be available for `gen_traffic.py` and `malicious_traffic.py` to run:

- Python 2.7
- python-scapy >=2.2.0

In Ubuntu:

```
1 $ sudo apt-get install python-libpcap python-scapy
```

E.2 Usage

E.2.1 Normal Traffic Generator.

`gen_traffic.py` generates random TCP and UDP packets with the given characteristics and rates. Table E.1 shows all the available options and the defaults where applicable. The generator can be stopped by pressing `Ctrl-c` or sending it `SIGTERM` via `kill`.

E.2.2 Validation.

Validation is done manually on this tool by capturing the generated traffic on both the sending and receiving hosts. Full packet logging may be done by the generator directly (raw packets bytes were dumped), so a simple comparison between the sent traffic, the

traffic it believes it generated, and the assigned settings ensure all parts of the tool operate as expected.

E.2.2.1 Examples.

Send UDP traffic to 192.168.1.5:2000 twice a second:

```
$ ./gen_traffic.py -t udp -p 2000 -h 192.168.1.5 -d .5
1356728008.79 LOG4 START: Starting at 28 Dec 2012 15:53:28
1356728008.79 LOG4 Starting a valid UDP sender to \
   192.168.1.5:2000
1356728008.79 LOG4 LOCAL ADDRESS: 192.168.1.115:2000
1356728009.29 LOG4 Sent valid \
   17:dae8053de4050a230b106d763665f058 to 192.168.1.5:2000
1356728009.29 LOG4 Received valid \
   17:f537a893140dbcc3b911cb69eae34b46 from 192.168.1.5:2000
1356728009.79 LOG4 Sent valid \
   17:500b1c56df2e3d17ce50bda4e9be03c9 to 192.168.1.5:2000
...
1356728011.81 LOG4 Sent valid \
   17:f31dd5564b658e956fe74cc35c0f603f to 192.168.1.5:2000
1356728011.81 LOG4 Received valid \
   17:ed8104a0ab9835fab38cc453362e8894 from 192.168.1.5:2000
^C1356728012.25 LOG4 User requested we stop
1356728012.25 LOG4 UDP sender to 192.168.1.5:2000 dying
```

Receive that UDP traffic:

```
$ ./gen_traffic.py -t udp -l -p 2000
1356727988.4 LOG4 START: Starting at 28 Dec 2012 15:53:08
1356727988.4 LOG4 Starting a UDP receiver on port 2000
1356727988.4 LOG4 LOCAL ADDRESS: 192.168.1.5:2000
```

```
1356728009.29 LOG4 Received valid \
   17:dae8053de4050a230b106d763665f058 from \
   192.168.1.115:49958
1356728009.29 LOG4 Sent valid \
   17:f537a893140dbcc3b911cb69eae34b46 to 192.168.1.115:49958
...
1356728011.81 LOG4 Received valid \
   17:f31dd5564b658e956fe74cc35c0f603f from \
   192.168.1.115:49958
1356728011.81 LOG4 Sent valid \
   17:ed8104a0ab9835fab38cc453362e8894 to 192.168.1.115:49958
^C1356728015.71 LOG4 User requested we stop
1356728015.71 LOG4 UDP listener on port 2000 dying
```

E.2.3 Malicious Traffic Generator.

`malicious_traffic.py` generates malicious ARG traffic by replaying ARG protocol traffic with randomly chosen modifications. The possible modifications are shown below. Each may be chosen with a 10% probability.

- Zero ARG signature/HMAC
- Change message type
- Zero data
- Remove the data
- Changed sequence number
- Changed source IP address
- Changed destination IP address

The only option available for `malicious_traffic.py` is `--output` option. This has the same effect as `gen_traffic.py`'s and is covered in Table E.1.

Table E.1: gen_traffic.py command-line parameters

Parameter	Description	Possible Values	Default
`--type`	Type of traffic to work with	tcp, udp	
`--is-invalid`	Log traffic as invalid. No effect on actual traffic.		false
`--host`	Host to send to	IP or domain name	
`--port`	Port to send to/listen on	Port number 0-65535	
`--listen`	Listen on given port rather than initiating connection		false
`--delay`	Delay in seconds between sends	>=0.0s	1.0
`--echo`	Echo received packet data back to sender rather than random		false
`--size`	Size of packet data to send	Number of bytes to send	Random
`--output`	Log to given file	file name	stdout

E.2.3.1 Validation.

Validation is done manually on this tool by capturing the generated traffic on both the sending and receiving hosts. The generator logs the modification(s) it makes to each packet, so a comparison between the received packet and the replayed packet quickly confirms each modification works as expected. Modifications are checked individually and in combination.

E.2.3.2 Limitations.

Due to the design of the test network, with a switch connecting the hosts, it is difficult for this tool to reliably sniff traffic (it does not attempt to ARP spoof or otherwise redirect traffic flow). To compensate for this the malicious generators are run on the gateways themselves, allowing them to see all of the traffic passing the gateway. A more realistic solution would be to use a true hub (the test network's switch did not allow this configuration) or sniff on a spanning port and send via a separate port.

Appendix F: Results Processor

After every test run, a custom utility processes the pcap and log files into a single database and extracts statistics from there. This appendix documents the usage of this tool and its operation. Section F.2 covers the required packages to process runs. Section F.3 covers usage of the results processor. Section F.4 documents the operation of the processor.

F.1 Validation

Validation is done manually on this tool via short, extremely low-traffic test runs. By performing tests with only a few packets in each direction, it is possible to manually ensure that all results the processor produces are accurate. These validation tests are done on everything from single-flow tests (one direction between just two hosts) and many-flow, multi-protocol tests.

F.2 Environment

The test processor runs on Ubuntu 12.04, Ubuntu 12.10, and Mac OSX 10.8. Other versions and distributions are untested, although they may work if the below requirements are met.

The following packages must be available for `process_run.py` to run:

- Python 2.7
- python-scapy >=2.2.0
- python-libpcap >=0.6.2

In Ubuntu:

```
1 $ sudo apt-get install python-libpcap python-scapy
```

F.3 Usage

`process_run.py` supports a variety of parameters, all of which are optional. The full parameter list and defaults are given in Table F.1. Full descriptions of each are given in the list below.

Table F.1: `process_run.py` command-line parameters

Parameter	**Short**	**Default**
`--help`	`-h`	
`--logdir`	`-l`	'.'
`--database`	`-db`	In-memory
`--empty-database`		false
`--skip-processing`		false
`--skip-stats`		false
`--offset`		0
`--start-offset`		0
`--end-offset`		0
`--show-cycles`		false
`--finish-indicator`		none

- `--help` - Display usage message.
- `--logdir`, `-l` - Path of directory with log and pcap files.
- `--database`, `-db` - Path to database of processed run data. Will be created if needed, otherwise existing data will be used. If the database is only partially processed, `process_run.py` will complete it. If not given, defaults to an entirely in-memory database.

- `--empty-database` - If given, all existing data in database is removed.
- `--skip-processing` - Do not attempt to process data, only produce results. If the database is incomplete, nothing will happen.
- `--skip-stats` - Do not calculate statistics after processing run data.
- `--offset` - Number of seconds to ignore at the beginning and end of a run.
- `--start-offset` - Number of seconds to ignore at beginning of run. Overrides value of `--offset`.
- `--end-offset` - Number of seconds to ignore at end of run. Overrides value of `--offset`.
- `--show-cycles` - If processing errors occur and the processor generates loops in the packet chains (i.e., following the next hop for each packet eventually would lead back around), show cycles may reveal the problem packet(s). Mostly obsolete.
- `--finish-indicator` - File to create after processing is complete. Intended for automation.

F.3.1 Example.

The most common use case, when the caller wants to process the run data in the current directory and create a database named `run.db` with the results:

```
~/results/basic-t0-l20-hr500ms-2012-11-24-08:54:21$ \
   process_run.py -db run.db
```

If a run takes several seconds to enter a steady state, it may be beneficial to ignore the first 30 seconds of the run. In addition, this example does its work from a different directory but leaves the results in the same location as the previous example:

```
~/results$ process_run.py -l \
   basic-t0-l20-hr500ms-2012-11-24-08:54:21 -db \
   basic-t0-l20-hr500ms-2012-11-24-08:54:21/run.db \
   --start-offset 30
```

F.4 Processor Execution

Run processing follows the steps below. It makes assumptions about the test network to determine where packets are headed and what hosts actually send and receive them, so logs must come from a network set up as documented in Figure 4.2.

1. Create database.
2. Read run settings from log files and file names.
3. Check settings for test setup problems, such as missing hosts.
4. Read through each PCAP file, entering every packet into the database with a hash of its data.
5. Read through each log file. For each send/receive/transformation (gateways passing a packet to/from their inside network) line:
   (a) Determine the single-hop source and destination of the packet (who sent it and which system should see it next).
   (b) Determine the true sender and receiver of packet. That is, who originally sent the packet and for whom it is intended.
   (c) Determine if it is intended to be a "valid" packet (i.e., should it reach its destination or not?).
   (d) Locate the packet in the database via hash and record the log information in the record.

(e) If it is a transformation packet (at a gateway), locate the send and receive packets and link them together.

6. Trace packets through the network, creating a chain of sends and receives. Packets that pass through the gateway follow the transformation through. That is, if the gateway receives a packet, alters it, and sends it out the other side, the packet sequence continues unbroken.

7. Check for packet cycles, which would indicate a tracing problem.

8. Copy true source and destination of each packet chain into all packets in the sequence. When a host sends a packet, it has an intended recipient. This information is copied into each packet in the chain, making it easier to look up.

9. Locate packet chain terminations. Each packet in a chain is given the ID number of the packet that ends the chain, allowing the processor to tell where each packet ended and if it reached the intended destination.

10. Produce statistics by querying the database for packets matching various criteria, such as packets that terminate at a different destination than intended.

Bibliography

[AA06] S. Antonatos and K. G. Anagnostakis. TAO: protecting against hitlist worms using transparent address obfuscation. In *Communications and Multimedia Security, 10th IPIP TC-6 TC-11 International Conference*, pages 12–21, Crete, Greece, Dec 2006.

[AAMA07] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis. Defending against hitlist worms using network address space randomization. *Computer Networking*, 51:3471–3490, August 2007.

[AN94] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 122–136, May 1994.

[AN95] R. Anderson and R. Needham. Robustness principles for public key protocols. In Don Coppersmith, editor, *Advances in Cryptology–CRYPT0 95*, volume 963 of *Lecture Notes in Computer Science*, pages 236–247. Springer Berlin Heidelberg, 1995.

[An01] J.H. An. Authenticated encryption in the public-key setting: Security notions and analyses. *http://eprint.iacr.org/2001/079*, 2001.

[APWJ03] M. Atighetchi, P. Pal, F. Webber, and C. Jones. Adaptive use of network-centric mechanisms in cyber-defense. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '03, Washington DC, 2003. IEEE Computer Society.

[ARMT06] M. Abu Rajab, F. Monrose, and A. Terzis. On the impact of dynamic addressing on malware propagation. In *Proceedings of the 4th ACM workshop on Recurring malcode*, WORM '06, pages 51–56, New York, NY, USA, 2006. ACM.

[AZEH09] F. Aloul, S. Zahidi, and W. El-Hajj. Two factor authentication using mobile phones. In *Computer Systems and Application. AICCSA 2009. IEEE/ACS International Conference on*, pages 641–644, may 2009.

[BBGR09] R. Benadjila, O. Billet, S. Gueron, and M. Robshaw. The Intel AES Instructions Set and the SHA-3 Candidates. In Mitsuru Matsui, editor, *Advances in Cryptology–ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 162–178. Springer Berlin Heidelberg, 2009.

[BN00] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Advances in Cryptology–ASIACRYPT 2000*, pages 531–545, 2000.

[Buc04] J.A. Buchmann. Cryptographic hash functions. *Introduction to Cryptography*, pages 235–248, 2004.

[CGKR] L. Colitti, S. H. Gunderson, E. Kline, and T. Refice. Evaluating IPv6 adoption in the Internet. In *PAM 2010*.

[CS03] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. pages 37–44, Aug 2003.

[Gue10] S. Gueron. Intel advanced encryption standard new instructions set. May 2010.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151.

[KFLD01] D. Kewley, R. Fink, J. Lowry, and M. Dean. Dynamic approaches to thwart adversary intelligence gathering. In *DARPA Information Survivability Conference Exposition II, 2001. DISCEX '01. Proceedings*, volume 1, pages 176–185, 2001.

[Mal97] G.S. Malkin. Dial-in virtual private networks using layer 3 tunneling. In *Local Computer Networks, 1997. Proceedings., 22nd Annual Conference on*, pages 555–561, 2-5 1997.

[MBH+05] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-Based One-Time Password Algorithm. RFC 4226 (Informational), December 2005.

[MG98] C. Madson and R. Glenn. The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404 (Proposed Standard), November 1998.

[MHCN96] A. Mauthe, D. Hutchison, G. Coulson, and S. Namuye. Multimedia group communications: towards new services. *Distributed Systems Engineering*, 3(3):197, 1996.

[MKR+04] G. Miklós, F. Kubinszky, A. Rácz, Z. Turányi, A. Valkó, M. A. Rónai, and S. Molnár. A novel scheme to interconnect multiple frequency hopping channels into an ad hoc network. *SIGMOBILE Mobile Computer Communication Rev.*, 8:109–124, January 2004.

[MMPR11] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-Based One-Time Password Algorithm. RFC 6238 (Informational), May 2011.

[MPS+02] J. Michalski, C. Price, E. Stanton, E. Lee, CHUA, K. Seah, Yip Heng, TAN, and C. Pheng. Final report for the network security mechanisms utilizing network address translation ldrd project. Technical Report SAND2002-3613, Sandia National Laboratories, Albuquerque, New Mexico, 11 2002.

[Nar04] T. Narten. Assigning Experimental and Testing Numbers Considered Useful. RFC 3692 (Best Current Practice), January 2004.

[NV09] A.A. Neto and M. Vieira. Untrustworthiness: A trust-based security metric. In *Risks and Security of Internet and Systems (CRiSIS), 2009 Fourth International Conference on*, pages 123–126, Oct. 2009.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[Ram99] B. Ramsdell. S/MIME Version 3 Message Specification. RFC 2633 (Proposed Standard), June 1999. Obsoleted by RFC 3851.

[Rep08] K. A. Repik. Defeating Adversary Network Intelligence Efforts with Active Cyber Defense Techniques. Master's thesis, Air Force Institute of Technology, Fairborn, OH, June 2008.

[RT10] B. Ramsdell and S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard), January 2010.

[Sha49] C.E. Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.

[SK02] R. Song and L. Korba. Review of network-based approaches for privacy. In *Proceedings of the 14th Annual Canadian Information Technology Security Symposium*. National Research Council Canada, May 2002.

[SSH05] M. Sifalakis, S. Schmid, and D. Hutchison. Network address hopping: a mechanism to enhance data protection for packet communications. In *ICC 2005. IEEE International Conference on Communications*, volume 3, pages 1518–152, May 2005.

[SWLX02] J. Sjoberg, M. Westerlund, A. Lakaniemi, and Q. Xie. Real-Time Transport Protocol (RTP) Payload Format and File Storage Format for the Adaptive Multi-Rate (AMR) and Adaptive Multi-Rate Wideband (AMR-WB) Audio Codecs. RFC 3267 (Proposed Standard), June 2002. Obsoleted by RFC 4867.

[Tal12] D. Talbot. A bandwidth breakthrough. *MIT Technology Review*, Oct 2012.

[WL03] W. Weinstein and J. Lepanto. Camouflage of network traffic to resist attack (CONTRA). In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 2, pages 126–127, april 2003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2013	Master's Thesis	Oct 2011–Mar 2013

4. TITLE AND SUBTITLE

Evaluating the Effectiveness of IP Hopping via an Address Routing Gateway

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Morehart, Ryan A., Second Lieutenant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-13-M-35

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

INTENTIONALLY LEFT BLANK

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This thesis explores the viability of using Internet Protocol (IP) address hopping in front of a network as a defensive measure. This research presents a custom gateway-based IP hopping solution called Address Routing Gateway (ARG) that acts as a transparent IP address hopping gateway. This thesis tests the overall stability of ARG, the accuracy of its classifications, the maximum throughput it can support, and the maximum rate at which it can change IPs and still communicate reliably.

This research is accomplished on a physical test network with nodes representing the types of hosts found on a typical, corporate-style network. Direct measurement is used to obtain all results for each factor level. Tests demonstrate ARG classifies traffic correctly, with no false negatives and less than a 0.15% false positive rate on average. The test environment conservatively shows this to be true as long as the IP address change interval exceeds two times the network's round-trip latency; real-world deployments may allow for more frequent hopping. Results show ARG capably handles traffic of at least four megabits per second with no impact on packet loss. Fuzz testing validates the stability of ARG itself, although additional packet loss of around 23% appears when under attack.

15. SUBJECT TERMS

Networks, address space randomization, IP hopping, routing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins
U	U	U	UU	116	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 ext. 7979